Computing Curriculum - Software Engineering

--- Public Draft 1 ---
(July 17, 2003)

This is a draft document distributed for purposes of review by the public (those interested in the education of software engineers). At this point, the document is not complete or authoritative; it is subject to revision; and it does not necessarily represent the contents of the final document!

The Joint Task Force on Computing Curricula
IEEE Computer Society
Association for Computing Machinery

This material is based upon work supported by the
National Science Foundation under Grant No. 0003263

Preface

This document was developed through an effort originally commissioned by the ACM Education Board and the IEEE-Computer Society Educational Activities Board to create curriculum recommendations in several computing disciplines: computer science, computer engineering, software engineering and information systems. Other professional societies have joined in a number of the individual projects. Such has notably been the case for the CCSE (Computing Curricula – Software Engineering) project, which has included participation by representatives from the Australian Computer Society, the British Computer Society, and the Information Processing Society of Japan.

Development Process

The CCSE project has been driven by a Steering Committee appointed by the sponsoring societies. The development process began with the appointment of the Steering Committee co-chairs and a number of the other participants in the fall of 2001. More committee members, including representatives from the other societies were added in the first half of 2002. The following are the members of the CCSE Steering Committee:

Co-Chairs

Rich LeBlanc, ACM, Georgia Institute of Technology, U.S.
Ann Sobel, IEEE-CS, Miami University, U.S.

Knowledge Area Chair

Ann Sobel, Miami University, U.S.

Pedagogy Focus Group Co-Chairs

Mordechai Ben-Menachem, Ben-Gurion University, Israel
Timothy C. Lethbridge, University of Ottawa, Canada

Co-Editors

Jorge L. Díaz-Herrera, Rochester Institute of Technology, U.S.
Thomas B. Hilburn, Embry-Riddle Aeronautical University, U.S.

Organizational Representatives

ACM: Andrew McGettrick, University of Strathclyde, U.K.
ACM SIGSOFT: Joanne M. Atlee, University of Waterloo, Canada
ACM Two-Year College Committee: Elizabeth Hawthorne, Union County College, U.S.
Australian Computer Society: John Leaney, University of Technology Sydney, Australia
British Computer Society: David Budgen, Keele University, U.K.
Information Processing Society of Japan: Yoshihiro Matsumoto, Musashi Institute of Technology, Japan
IEEE Computer Technical Committee on Software Engineering: Barrie Thompson, University of Sunderland, U.K.

The construction of this volume has centered around two major efforts that have engaged a large number of volunteers, as well as all of the members of the Steering Committee. The first of these efforts was development of a set of desired curriculum outcomes and a statement of what every SE graduate should know. These ideas are captured in our statement of required Software Engineering Education Knowledge (SEEK), presented in Chapter 5 of this document. The second effort was the construction of a set of curriculum recommendations, describing how a software engineering curriculum incorporating the material from the SEEK can be structured in various contexts. These are presented in Chapter 6 of this document.

Work began on SEEK in Spring 2002 with the involvement of nine groups of volunteers, leading to an NSF-supported workshop in June 2002 where representatives of the volunteer groups met with some Steering Committee members, resulting in the first "internal" draft of the SEEK. This draft was reviewed by all of the Steering Committee and a group of outside software engineering "experts"; revised by the Steering Committee based on comments from this reviews; and then published for public comment in August 2002. Comments from these public reviews were used to create a second draft by December 2002.

Six "pedagogy focus groups" were created in November 2002 to begin the process of developing the curriculum recommendations. Each of these groups consisted of committee of volunteers plus one or two Steering Committee members. Input by these groups and further work by some members of the Steering Committee resulted in an initial curriculum draft in March 2003. This draft was discussed at a workshop at the Conference on Software Engineering Education and Training held that month in Madrid, Spain and with members of the Working Group on Software Engineering Education and Training at their meeting just before the conference. Feedback from these discussions was used to revise the draft in preparation for publishing it for public review in May 2003, along with a draft of the rest of this volume.

The first public review of the draft was at the Summit on Software Engineering Education held at the International Conference of Software Engineering in Portland, Oregon, early in May 2003.

Acknowledgements

The development of this document was support by the National Science Foundation, the Association of Computing Machinery, and the IEEE Computer Society.

Since its inception, many individuals have contributed to the CCSE project, some in more than one capacity. This work could not have been completed without the dedication and expertise of these volunteers. Appendix C lists the names of those that have participated in the various development and review stages of this document. Special thanks go to Susan Mengel of Texas Tech University who served as an original co-chair of the Steering Committee and performed the initial organization tasks for the CCSE project.

Table of Contents

Chapter 1: Introduction

1.1 Purpose of this volume

The primary purpose of this volume is to provide guidance to academic institutions and accreditation agencies about what should constitute an undergraduate software engineering education. These recommendations have been developed by a broad, internationally-based group of volunteer participants. This group has taken into account much work that has been done in software engineering education over the last quarter of a century. Software engineering curriculum recommendations are of particular relevance, since there is currently a surge in the creation of software engineering degree programs and an accreditation process for such programs has been established in a number of countries.

The recommendations included in this volume are based on a high-level set of characteristics of software engineering graduates presented in Chapter 2. Flowing from these outcomes are the two main contributions of this document:

- SEEK: Software Engineering Education Knowledge - what every SE graduate must know.
- Curriculum: Ways that this knowledge and the skills fundamental to software engineering can be taught in various contexts.

1.2 Where we fit in the CC picture

In 1998, the Association for Computing Machinery (ACM) and the Computer Society of the Institute for Electrical and Electronic Engineers (IEEE-CS) convened a joint curriculum task force called *Computing Curricula 2001,* or *CC2001* for short. In its original charge, the CC2001 Task Force was asked to develop a set of curricular guidelines that would “match the latest developments of computing technologies in the past decade and endure through the next decade.” This task force came to recognize early in the process that they—as a group primarily composed of computer scientists—were ill-equipped to produce guidelines that would cover computing technologies in their entirety. Over the past fifty years, *computing* has become an extremely broad designation that extends well beyond the boundaries of computer science to encompass such independent disciplines as computer engineering, software engineering, information systems, and many others. Given the breadth of that domain, the curriculum task force concluded that no group representing a single specialty could hope to do justice to computing as a whole. At the same time, feedback they received on their initial draft made it clear that the computing education community strongly favored a report that did take into account the breadth of the discipline.

Their solution to this challenge was to continue their work on the development of a volume of computer science curriculum recommendations, published in 2001 as the *CC2001 Computer Science* volume (CCCS volume)[ACM 2001b]. In addition, they recommended to their sponsoring organizations that the project be broadened to include volumes of recommendations for the related disciplines listed above, as well as any others that might be deemed appropriate by the computing education community. This volume represents the work of the CCSE (Computing

Curricula – Software Engineering) project and is the first such effort by the ACM and the IEEE-CS to develop curriculum guidelines for software engineering.

In late 2002, *IS 2002 - Model Curriculum and Guidelines for Undergraduate Degree Programs in Information Systems* was approved and published, having been created by a task force chartered by the ACM, the Association for Information Systems (AIS) and the Association of Information Technology Professionals (AITP). Additional efforts are ongoing to produce recommended curricula for software engineering (this volume), computer engineering, and information technology.

1.2.1 Computer Science volume

Because computer science provides some of the scientific underpinnings of software engineering, the computer science volume plays a special role in relation to this software engineering volume. In Chapter 5, the SEEK includes specific reference to core topics described in the CCCS volume. Additionally, among the curriculum structure alternatives presented in Chapter 6 are some that include use of particular courses described in the computer science volume.

1.3 Structure of the volume

Chapter 2 presents the guiding principles behind the development of this document. These principles were adapted from those originally articulated by the CC2001 Task Force as they began work on what became the CCCS volume. Chapter 3 describes some of the history of software engineering education and how it has influenced the recommendations in this document. Chapter 4 discusses models of curriculum structure that form the basis of the particular recommendations presented here. Chapter 5 provides the description of what every SE graduate should know, the body of Software Engineering Education Knowledge (the SEEK) that underlies the curriculum designs presented in Chapter 6. Finally, Chapter 7 addresses a variety of curriculum implement challenges and also considers assessment approaches.

Chapter 2: Guiding Principles

This chapter describes the foundational ideas and beliefs that guided the development of the CCSE materials: the guiding principles for the entire CCSE effort, and the desired student outcomes for an undergraduate curriculum in software engineering.

2.1 CCSE Principles

This section describes the foundational ideas and beliefs that guided the development of the CCSE materials. The following list of principles were strongly influenced by the principles set down in the CCCS volume; in some cases they are minor rewording of the those principles. For others, we have tried to capture the special nature of software engineering that differentiates it from other computing disciplines.

[1] *Computing is a broad field that extends well beyond the boundaries of any one computing discipline*. CCSE concentrates on the knowledge and pedagogy associated with a software engineering curriculum. Where appropriate, it will share or overlap with material contained in other Computing Curriculum reports and will offer guidance on its incorporation into other disciplines.

[2] *Software Engineering draws its foundations from a wide variety of disciplines*. Undergraduate study of software engineering relies on many areas in computer science for its theoretical and conceptual foundations, but it also requires students to utilize concepts from a variety of other fields, such as mathematics, engineering and project management. All software engineering students must learn to integrate theory and practice, to recognize the importance of abstraction and modeling, to be able to acquire special domain knowledge beyond the computing discipline for the purposes of supporting software development in specific domains of application, and to appreciate the value of good engineering design.

[3] *The rapid evolution and the professional nature of software engineering require an ongoing review of the corresponding curriculum.* The 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 engineers to the public, it is important that the curriculum guidance support and promote effective external assessment and accreditation of software engineering programs.

[4] *Development of a software engineering curriculum must be sensitive to changes in technology, new developments in pedagogy, and the importance of lifelong learning.* In a field that evolves as rapidly as software engineering, educational institutions must adopt explicit strategies for responding to change. Institutions, for example, must recognize the importance of remaining abreast of well-established progress in both technology and pedagogy, subject to the constraints of available resources. Software engineering education, moreover, must seek to prepare students for lifelong learning that will enable them to move beyond today's technology to meet the challenges of the future.

[5] *CCSE must go beyond knowledge elements to offer significant guidance in terms of individual curriculum components*. The CCSE curriculum models should assemble the knowledge elements into reasonable, easily implemented learning units. Articulating a set of

well-defined models will make it easier for institutions to share pedagogical strategies and tools. It will also provide a framework for publishers who provide the textbooks and other materials.

[6] *CCSE must support the identification of the fundamental skills and knowledge that all software engineering graduates must possess.* Where appropriate, CCSE must help define the common themes of the discipline and ensure that all undergraduate program recommendations include this material.

[7] *Guidance on software engineering curricula must be based on an appropriate definition of software engineering knowledge*. The description of this knowledge should be concise, appropriate for undergraduate education, and it should use the work of previous studies on the software engineering body of knowledge. A core set of required topics, from this description, must be specified for all undergraduate software engineering degrees. The core should have broad acceptance by the software engineering education community. Coverage of the core will start with the introductory courses, extend throughout the curriculum, and be supplemented by additional courses that may vary by institution, degree program, or individual student.

[8] *CCSE must strive to be international in scope.* Despite the fact that curricular requirements differ from country to country, CCSE is intended to be useful to computing educators throughout the world. Where appropriate, every effort is being made to ensure that the curriculum recommendations are sensitive to national and cultural differences so that they will be widely applicable throughout the world. The involvement by national computing societies and volunteers from all countries will be actively sought and welcomed.

[9] *The development of CCSE must be broadly based*. To be successful, the process of creating software engineering education recommendations must include participation from the many perspectives represented by software engineering educators and by industry, commerce, and government professionals.

[10] *CCSE must include exposure to aspects of professional practice as an integral component of the undergraduate curriculum*. The education of all software engineering students must include student experiences with the professional practice of software engineering. The professional practice of software engineering encompasses a wide range of issues and activities including problem solving, management, ethical and legal concerns, written and oral communication, working as part of a team, and remaining current in a rapidly changing discipline.

[11] *CCSE must include discussions of strategies and tactics for implementation, along with high-level recommendations*. Although it is important for CCSE to articulate a broad vision of software engineering education, the success of any curriculum depends heavily on implementation details. CCSE must provide institutions with advice on the practical concerns of setting up a curriculum.

2.2 Student Outcomes

As a first step in providing curriculum guidance, the following set of outcomes for an undergraduate curriculum was developed. This is intended as a generic list that could be adapted to a variety of software engineering program implementations.

Graduates of an undergraduate SE program must be able to:

[1] Show mastery of the necessary body of knowledge and skills to begin practice as a software engineer.

[2] Work as an individual and as part of a team to develop and deliver executable artifacts.

[3] Reconcile conflicting objectives, finding acceptable compromises within limitations of cost, time, knowledge, existing systems, and organizations.

[4] Design appropriate solutions in one or more application domains using engineering approaches that integrate ethical, social, legal, and economic concerns.

[5] Demonstrate an understanding of and apply current theories, models, and techniques that provide a basis for problem identification and analysis, software design, development, implementation and verification.

[6] Negotiate, work effectively, provide leadership where necessary, and communicate well with stakeholders in a typical software development environment.

[7] Learn new models, techniques, and technologies as they emerge and appreciate the necessity of such continuing professional development.

Chapter 3: The Software Engineering Discipline

This chapter discusses the nature of software engineering and some of the history and background that is relevant to the development of software engineering curriculum guidance. The purpose of the chapter is to provide context and rationale for the curriculum materials in subsequent chapters.

3.1 The Discipline of Software Engineering

Since the dawn of computing in the 1940s, the applications and use of computers have grown at a staggering rate. Software plays a central role in almost all aspects of daily life: in government, banking and finance, education, transportation, entertainment, medicine, agriculture, and law. The number, size, and application domains of programs has grown dramatically; as a result, billions are being spent on software development, and the livelihood and lives of millions directly depend on the effectiveness of this development. Software products have helped us to be more efficient and productive; they make us more effective problem solvers; and they provide us with an environment for work and play that is safer, more flexible, and less confining. Despite these successes, there are serious problems in the cost, timeliness, and quality of many software products. The reasons for these problems are many-fold:

- Software products are some of the most complex of man-made systems and software, by its very nature, has intrinsic difficulties (e.g., complexity, visibility, and changeability) that are not easily overcome [Brooks 95].
- Programming techniques and processes that worked effectively in the 1950s and early 1960s, to develop modest-sized programs by an individual or a small team, have not scaled-up well to the development of large, complex systems (systems with millions of lines of code, requiring years of the work, by hundreds of engineers).
- The pace of change in computer and software technology drives the demand for new and evolved software products. Our successes in this area have created customer expectations and completive forces that stress the quality of software and their development schedules.

It has been over thirty years since the first organized, formal discussion of software engineering as a discipline took place at the 1968 NATO Conference on Software Engineering [Naur 1969]. The term “software engineering” is now widely used in industry, government and academia: thousands of computing professionals go by the title “software engineer”; numerous publications, groups and organizations, and professional conferences use the term software engineering in their names; and there are many educational courses and programs on software engineering. However, there are still disagreements and differences about the meaning of the term. The following definitions depict a variety of descriptions about the meaning and nature of software engineering. However, they all possess a common thread, which says, or strongly implies: software engineering is more than just coding; it includes concerns about quality, schedule and cost; and to be successful, a software engineer needs discipline, knowledge, and professional experience.

Definitions of Software Engineering:

- "The establishment and use of sound engineering principles (methods) in order to obtain economically software that is reliable and works on real machines" [Bauer 1972].
- "The application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software" [IEEE 1990].
- "… the technological and managerial discipline concerned with systematic production and maintenance of software products that are developed and modified on time and within cost estimates" [Fairley 1985].
- "… the computer science discipline concerned with developing large applications. Software engineering covers not only the technical aspects of building software systems, but also management issues, such as directing programming teams, scheduling, and budgeting" (WebReference Webopaedia).
- SEI software engineering definition from 1990 SEI Report on Undergraduate Software Engineering Education (CMU/SEI-90-TR-003):
  - "Engineering is the systematic application of scientific knowledge in creating and building cost-effective solutions to practical problems in the service of mankind."
  - "Software engineering is that form of engineering that applies the principles of computer science and mathematics to achieving cost-effective solutions to software problems."
- "Software engineering applies engineering discipline to software development, ensuring that software products will meet organizational, financial, marketplace, and technical requirements. Like other fields of engineering, software engineering is a hybrid of scientific, technical and management principles ... In short, software engineering is the engineering of software." (http://www.omse.org/whatis.htm)

A central point in these definitions is that the creation of software is essentially an engineering The study and practice oriented discipline. It is about creating high-quality software in a systematic, controlled and efficient manner. As such, there are important emphases on analysis and evaluation, specification, design, implementation and evolution of software. In addition, there are issues related to quality, to novelty and creativity, to individual skills, and to teamwork and professional practice that play a vital role in software engineering.

3.2 An Engineering Discipline

The study and practice of software engineering, as with other engineering disciplines, is influenced by the general nature of engineering as an academic discipline and as a profession. Engineering disciplines have emerged from ad hoc practice by the exploitation and management of technology, and by the application of maturing science. The professional engineer possesses knowledge of mathematics and science, and through the principles of analysis and design applies this knowledge in a judicious way to utilize materials in the solution of problems and development of products for the benefit of mankind.

3.2.1 Characteristics of Engineering

There are a set of features that not only are common to every engineering discipline, but are so predominant and critical that they can be used to describe the underpinnings of the engineering discipline, and in particular that of software engineering. The following is a list of characteristics of engineering and engineers that has influenced the development of software engineering, and this volume:

[1] Engineering is about solving customer problems. Because of the pervasive nature of software, the scope for the types of problems in SE may be significantly wider than in other branches of engineering.

[2] Engineers proceed by making a series of decisions, carefully evaluating options, and choosing an approach at each decision-point that is appropriate for the current task in the current context. Appropriateness can be judged by tradeoff analysis, which balances costs against benefits. The current context can dramatically affect the decision made; for example, a safety-critical application will require quite different decisions from a system where safety is not a concern.

[3] Engineers measure things and work quantitatively when appropriate. They also calibrate and validate their measurements.

[4] Engineering is a creative discipline: the ability to design in a proficient manner is a hallmark of a good engineer. Engineers concentrate their efforts on problem analysis and solution (design).

[5] Engineers take on many different roles: Generally accepted functions that engineers can perform include research, development, design, production, testing, construction, operations, management, and others such as sales, consulting and teaching. All of the engineering functions have their counterpart in software engineering; they are all well defined within a specific process for applying engineering design for software. Engineered products range from devices and systems to processes and structures.

[6] Engineers must apply knowledge from other disciplines –in addition to their own, in particular mathematics, basic sciences and economics. In software engineering, underlying disciplines of central importance are computer science, discrete mathematics and psychology. Disciplines such as physics and continuous mathematics support some applications of software engineering, but are less central to software engineering itself than they are to other branches of engineering.

[7] Choice and use of appropriate tools is key to engineering. Engineers also create tools and this is more prominent in software engineering since their tools are formalisms directly supported, in most cases, by software systems.

[8] Engineering disciplines advance by the development and validation of principles and best practices. The software engineering principles are a specialized subset of general engineering principles. These principles motivate the creation of software engineering standards; whose detailed implementation is viewed as best practices; this is illustrated in the Figure 1.

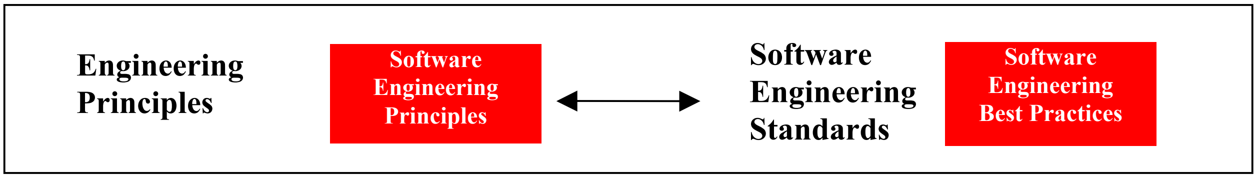


Figure 1: Relationship of Principles and Practice

[9] Knowledge about and ability to reuse existing engineering artifacts are important in advancing engineering productivity and quality. This is particularly relevant within a specific domain of discourse.

[10] Engineers learn to work in a disciplined and systematic manner. The process that engineers follow must adapt to the appropriate context.

[11] Engineers work in teams with other engineers and with other professionals; this leads to the need for them to develop communications and teamwork skills, and for them to know when to consult others, when they lack knowledge.

[12] Engineering is a profession; hence engineers follow ethical and professional principles to protect society, their customers, their employers and themselves.

[13] Engineers must continue to update their knowledge about new methods, techniques and technology.

3.2.2 Engineering design

Engineering design is central to any engineering activity, and it plays a critical role in software. However, software engineering goes beyond traditional engineering design and includes "implementation" activities found in traditional "manufacturing." Furthermore, continued evolution (i.e., "maintenance") is also of more critical importance for software.

In general, engineering design activities refer to the definition of a new product by finding technical solutions to specific practical issues, while taking into account economic, legal, and ecological considerations. As such, engineering design provides the prerequisites for the physical realization of a solution by following a systematic process that best satisfy the requirements within potentially conflicting constraints. This process typically follows a step-wise approach from problem formulation and analysis, prototyping and evaluation, and decision and production, all under a system view of the major phases; these phases include planning, preliminary study or operational concept, design, development, installation, operation (and maintenance), and retirement. This process is remarkably similar to what in the software engineering community is known as the software life cycle [Royce 1970]. From this point of view, the *process of software development* corresponds to what is generally known as engineering design.

3.2.3 Domain-specific software engineering

Within a specific domain, the engineer relies on specific education and experience to evaluate many possible solutions, keeping in mind cost of manufacture, ease of production, availability of materials, performance requirements, etc. Engineers have to determine which standard parts can be used and which parts have to be developed from scratch. To make the necessary decisions, they must have a fundamental knowledge of specialty subjects as well as an understanding of economics and people.

While domains span the entire spectrum of industry, government, and society, in this volume, we point to a smaller list of specialty application areas (see section 5.18). We feel that graduates of software engineering programs should be able to produce software that is a genuine value to problems in a particular domain; they should come to terms with at least the fundamentals of one application domain.

3.3 Professional Practice

A key objective of any engineering program would be to provide graduates with the tools necessary to begin the professional practice of engineering. Principle 10, in Chapter 2, states: "The education of all software engineering students must include student experiences with the professional practice of software engineering". The content and nature of such experiences are discussed in subsequent chapters, while this section provides rationale and background for the inclusion of professional practice elements in a software engineering curriculum.

3.3.1 Rationale

All of the characteristics of engineering discussed in Section 3.2.1 relate, directly or indirectly, to the professional practice of engineers. Those most directly relevant to professional practice speak to the need for "communications and teamwork skills", "ethical and professional principles", "engineering productivity and quality", "work in a disciplined and systematic manner" and engineers to continue to "update their knowledge about new methods, techniques and technology". Employers of engineering program graduates often speak to these same needs [Denning 1992]. Each year, the National Association of Colleges and Employers conducts a survey to determine what qualities employers consider most important in applicants seeking employment [NACE 2003]. In 2003, employers were asked to rate the importance of candidate qualities and skills on a five-point scale, with five being "extremely important" and one being "not important." Communication skills (4.7 average), honesty/integrity (4.7), teamwork skills (4.6), interpersonal skills (4.5), motivation/initiative (4.5), and strong work ethic (4.5) were the most desired characteristics.

The dual challenges of society's critical dependence on the quality and cost of software, and the relative immaturity of software engineering, makes attention to professional practice issues even more important to software engineering programs than many other engineering programs. Graduates of software engineering programs need to arrive in the workplace equipped to meet these challenges and to help to evolve the software engineering discipline into a more professional and accepted state.

3.3.2 Software Engineering Code of Ethics and Professional Practices

Software Engineering as a profession has obligations to society. The products produced by software engineers affect the lives and livelihood of the clients and users of those products. Hence, software engineers need to act in an ethical and professional manner. The preamble to the *Software Engineering Code of Ethics and Professional Practice* [ACM 1998] states

> "Because of their roles in developing software systems, software engineers have significant opportunities to do good or cause harm, to enable others to do good or

> cause harm, or to influence others to do good or cause harm. To ensure, as much as possible, that their efforts will be used for good, software engineers must commit themselves to making software engineering a beneficial and respected profession. In accordance with that commitment, software engineers shall adhere to the following Code of Ethics and Professional Practice."

In order to help insure ethical and professional behavior software engineering educators have an obligation to not only make their students familiar with the *Code*, they must find ways for students to engage in discussion and activities that illustrate and illuminate its eight principles. In Chapter 4, this area is included as part of the expected student outcomes of a software engineering curriculum.

3.3.3 Curriculum Support for Professional Practice

A curriculum can have an important direct affect on some of professional practice factors (e.g., teamwork, communication, and analytic skills), while others (e.g. strong work ethic, self-confidence) are subject to the more subtle influence of a college education on individual's character, personality and maturity. In this volume elements of professional practice that should be part of any curriculum are identified in Chapter 5, and Chapters 6 and 7 contain guidance and ideas for incorporating material about professional practice in a software engineering curriculum. In particular, there is consideration of material directly supportive of professional practice (technical communications, ethics, engineering economics, etc.) and ideas about the modeling of work environments (case studies, laboratory work, team project courses).

There are many elements, outside the classroom, that can have a significant affect on a student's preparation for professional practice. The following are some examples: involvement in the core curriculum of faculty who have professional experience; student work experience as an intern or as part of a cooperative education program; and extracurricular activities such as attending colloquia, field trips visits to industry, and participating in student professional clubs and activities.

3.4 Prior Software Engineering Education and Computing Curriculum Efforts

In the late 1980s and early 1990s, software engineering education was fostered and supported by the efforts of the Education Group of the Software Engineering Institute (SEI), at Carnegie Mellon University. These efforts included the following: surveying and reporting on the state of software engineering education; publishing curriculum recommendations for graduate software engineering programs; organizing and facilitating workshops for software engineering educators; and publishing software education curriculum modules.

The SEI initiated and sponsored the first Conference on Software Engineering Education and Training (CSEET), held in 1987. The CSEET has since provided a forum for SE educators to meet, present and discuss SE education issues, methods, and activities. In 1995, as part of its education program, the SEI started the Working Group on Software Engineering Education and Training (WGSEET) (http://www.sei.cmu.edu/collaborating/ed/workgroup-ed.html). The WGSEET objective is to investigate issues, propose solutions and actions, and share information and best practices with the software engineering education and training community. In 1999, the Working Group produced a technical report offering guidelines on the design and implementation of undergraduate software engineering programs [Bagert 1999].

In 1993, the IEEE-CS and the ACM established the IEEE-CS/ACM Joint Steering Committee for the Establishment of Software Engineering as a Profession. Subsequently, the Steering committee was replaced by the Software Engineering Coordinating Committee (SWECC), which coordinated the work of three efforts: the development of a Code of Ethics and Professional Practices [ACM 1998], the Software Engineering Education Project (SWEEP) that developed a draft accreditation criteria for undergraduate programs in software engineering [Barnes 1998], and the development of a *Guide to the Software Engineering Body of Knowledge* (SWEBOK) [Bourque 2001]. All these efforts have influenced the philosophy and the content of this volume.

A major influence on the CCSE efforts has been the Curriculum 1991 report [Tucker 1991] and the CCCS volume [ACM 2001]. Elements, features, and ideas from these documents were used or adapted for use in this volume. In particular, the organization of this volume and the CCSE principles in Chapter 2 were strongly influenced by the 2001 computer science volume.

3.5 SWEBOK and other BOK Efforts

A major challenge in providing curriculum guidance for new and emerging, or dynamic disciplines is the identification and specification of the underlying content of the discipline. Since computing disciplines (computer engineering, computer science, information science, information technology, and software engineering) are both relatively new and dynamic, the specification of a "body of knowledge" is critical.

In Chapter 5 a body of knowledge is specified that supports software engineering education curricula (called SEEK - Software Engineering Education Knowledge). The organization and content was influenced by a number of previous efforts at describing the knowledge that comes from other related disciplines. The following is a description of such efforts:

- The PMBOK (*Guide to the Project Management Body of Knowledge*) [PMI 2000] provides a description of knowledge about project management (not limited to software projects). Besides its relevance to software project management, the PMBOK's organization and style has influenced similar, subsequent efforts in the computing disciplines.
- The IS'97 report (*Model Curriculum and Guidelines for Undergraduate Degree Programs in Information Systems*) [Davis, 1997] describes a model curriculum for undergraduate degree programs in Information Systems. The document includes a description of an IS body of knowledge (which included SE knowledge) and also a metric (similar to Bloom's levels in [Bloom 1956]) for prescribing the required depth of knowledge for undergraduates.
- The report "Computing as a Discipline" [ACM 1989] provides a comprehensive definition of computing and formed the basis for the work on Computing Curriculum 1991, and its successor Computing Curriculum 2001. It specifies nine subject areas that cover the computing discipline.
- The *Guidelines for Software Engineering Education* [Bagert 1999] (developed by the WGSEET), describes a curriculum model for undergraduate SE education that is based on a

body of knowledge consisting of four areas: Foundations, Core, Recurring and Support. These areas were further divided into components.

- The SWEBOK is a comprehensive description of the knowledge needed for the practice of software engineering. In addition to describing the knowledge needed to be a software engineer, one of the objectives of this project was to "Provide a foundation for curriculum development ...". To support this objective, the SWEBOK includes a rating system for its knowledge topics based on Bloom's levels of educational objectives. Although the SWEBOK was one of the primary sources used in the development of this document and there has been close communication between the SWEBOK and CCSE projects, there were assumptions and features of the SWEBOK that differentiate the two efforts:
  - the SWEBOK is intended to cover knowledge after four years of practice;
  - the SWEBOK intentionally does not cover non-SE knowledge that a software engineer must know;
  - and the CCSE is intended to only support undergraduate software engineering education.

3.6 Accreditation Development

In order to ensure consistent quality among programs and to promote and support their improvement, accreditation organizations are formed to provide accreditation policy and procedures to evaluate programs for accreditation purposes. The CCSE has attempted to avoid any conflict with existing software engineering accreditation policies and procedures. In particular, volunteers have been recruited for the CCSE effort who are familiar with accreditation requirements in various countries and regions (Australia, Canada, Europe, Japan, U.S.). On the other hand, it is expected that the CCSE recommendations and guidelines will influence the future direction and nature of software engineering accreditation. There is further discussion of accreditation and assessment issues in Chapter 9.

Chapter 4: Overview of Software Engineering Education Knowledge

4.1 Process of Determining the SEEK

The development model chosen for determining CCSE was based on the model used to construct the CCCS volume. Development of the CCSE volume has been divided into two groups: an Education Knowledge Area Group and a Pedagogy Focus Group. The education knowledge area group is responsible for defining and documenting a software engineering education body of knowledge appropriate for guiding the development of undergraduate software engineering curricula (see Appendix B for list). This body of knowledge is called Software Engineering Education Knowledge or SEEK. The pedagogy focus group is responsible for using SEEK to formulate guidance for pedagogy as well as course and curriculum design to support undergraduate software engineering degree programs.

The initial selection of the SEEK areas was based on the SWEBOK knowledge areas and multiple discussions with dozens of SEEK area volunteers. The SEEK area volunteers were divided into groups representing each individual SEEK area where each group contained roughly seven volunteers. These groups were assigned the task of providing the details of the units that compose a particular educational knowledge area and the further refinement of these units into topics. To facilitate their work, references to existing related software engineering body of knowledge efforts (e.g. SWEBOK, CSDP Exam, and SEI curriculum recommendations) and a set of templates for supporting the generation of units and topics were provided.

After the volunteer groups generated an initial draft of their individual education knowledge area details, the steering committee held a face-to-face forum that brought together education knowledge and pedagogy area volunteers to iterate over the individual drafts and generate an initial draft of the SEEK (see Appendix C for attendee list). This workshop held with this particular goal mirrored a similar overwhelmingly successful workshop held by CCCS at this very point in their development process. Once the content of the education knowledge areas were stabilized, topics were identified to be core or elective. Topics were also labeled with one of three Bloom's taxonomy's levels of educational objectives; namely, knowledge, comprehension, or application. Only these three levels of learning were chosen from Bloom's taxonomy since they represent what knowledge may be reasonably learned during an undergraduate education.

The workshop resulted in a complete internal draft of SEEK. The steering committee then arranged for a review of the internal draft by selected experts in the field, the advisory industrial council, and the knowledge area volunteers (see Appendix D for list). After this review was complete, the steering committee studied all reviewer comments and used them to revise the internal draft version of the SEEK. This work resulted in a public draft version of the SEEK. The steering committee has made this version of the SEEK available to the public and is soliciting reviews of it by those interested in undergraduate software engineering education.

After the completion of the public reviews of this document, the steering committee iterated over the reviewer comments to further refine and improve the contents of the SEEK. The public draft version was used at the start of the development of pedagogy, courses, and curricula. The final version was included in the first draft version of the CCSE Volume.

4.2 Knowledge Areas, Units, and Topics

Knowledge is a term used to describe the whole spectrum of content for the discipline: information, terminology, artifacts, data, roles, methods, models, procedures, techniques, practices, processes, and literature. The SEEK is organized hierarchically into three levels. The highest level of the hierarchy is the education knowledge **area,** representing a particular sub-discipline of software engineering that is generally recognized as a significant part of the body of software engineering knowledge that an undergraduate should know. Knowledge areas are high-level structural elements used for organizing, classifying, and describing software engineering knowledge. Each area is identified by an abbreviation, such as PRF for professional practices and is represented in this document with the color orange. Each area is broken down into smaller divisions called **units,** which represent individual thematic modules within an area. Adding a two or three letter suffix to the area identifies each unit; as an example, PRF.com is a unit on communication skills. Units are represented in this document with the color yellow. Each unit is further subdivided into a set of **topics,** which are the lowest level of the hierarchy. Topics are represented with either the color teal or white.

4.3 Core Material

In determining the SEEK, the steering committee recognizes that software engineering, as a discipline, is relatively young in its maturation and common agreement on definition of an education body of knowledge is evolving. The SEEK developed and presented in this document is based on a variety of previous studies and commentaries on the recommended content for the discipline. It was specially designed to support the development of undergraduate software engineering curricula, and therefore, does not include all the knowledge that would exist in a more generalized body of knowledge representation. The steering committee has therefore sought to define a **core** consisting of the essential material that professionals teaching software engineering agree is necessary for anyone to obtain an undergraduate degree in this field. By insisting on a broad consensus in the definition of the core, the steering committee hopes to keep the core as small as possible, giving institutions the freedom to tailor the elective components of the curriculum in ways that meet their individual needs. Material offered as part of an undergraduate program that falls outside the core is considered to be **elective.** Core topics are represented with the color teal and elective topics are represented with no color (white).

The following points should be emphasized to clarify the relationship between the SEEK and the steering committee's ultimate goal of providing undergraduate software engineering curriculum recommendations.

- *The core is not a complete curriculum.* Because the core is defined as minimal, it does not, by itself, constitute a complete undergraduate curriculum. Every undergraduate program must include additional elective units from the body of knowledge, although this document does not define what those units will be.

- *Core units are not necessarily limited to a set of introductory courses taken early in the undergraduate curriculum.* Although many of the units defined as core are indeed introductory, there are also some core units that clearly must be covered only after students have developed significant background in the field. For example, topics in such areas as project management, requirements elicitation, and abstract high-level modeling may require knowledge and sophistication that lower-division students do not possess. Similarly, introductory courses may include elective units alongside the coverage of core material. The designation *core* simply means *required* and says nothing about the level of the course in which it appears.

4.4 Unit of Time

The SEEK must define a metric that establishes a standard of measurement in order to judge the actual amount of time required to cover a particular unit. Choosing such a metric was quite difficult for the steering committee because no standard measure is recognized throughout the world. For consistency with the earlier curriculum reports, namely the other related computing curricula volumes to this effort, the task force has chosen to express time in **hours**. An hour corresponds to the actual in-class time required to present the material in a traditional lecture-oriented format (referred to in this document as contact hours). To dispel any potential confusion, however, it is important to underscore the following observations about the use of lecture hours as a measure:

- *The steering committee does not seek to endorse the lecture format.* Even though we have used a metric that has its roots in a classical, lecture-oriented format, the steering committee believes that there are other styles—particular given recent improvements in educational technology—that can be at least as effective. For some of these styles, the notion of hours may be difficult to apply. Even so, the time specifications should at least serve as a comparative measure, in the sense that a 5-hour unit will presumably take roughly five times as much time to cover as a 1-hour unit, independent of the teaching style.
- *The hours specified do not include time spent outside of class.* The time assigned to a unit does not include the instructor's preparation time or the time students spend outside of class. As a general guideline, the amount of out-of-class work is approximately three times the in-hours (3 in class and 9 outside).
- *The hours listed for a unit represent a minimum level of coverage.* The time measurements assigned for each unit should be interpreted as the *minimum* amount of time necessary to enable a student to perform the learning objectives for that unit. It is always appropriate to spend more time on a unit than the mandated minimum.

4.5 Relationship of the SEEK to the Curriculum

The SEEK does not represent the curriculum, but rather provides the foundation for the design, implementation and delivery of the educational units that make up a software engineering curriculum. Other chapters of the CCSE Volume provide guidance and support on how to use the SEEK to develop a curriculum. In particular, the organization and content of the knowledge areas and knowledge units should not be deemed to imply how the knowledge should be organized into education units or activities. For example, the SEEK does not advocate a sequential ordering of the KAs (1st CMP, 2nd FND, 3rd PRF, etc.). Nor does it suggest how topics and units should be combined into education units. Furthermore, the SEEK is not intended to purport any special curriculum development methodology (waterfall, incremental, cyclic, etc.).

4.6 Selection of Knowledge Areas

The initial selection of the SEEK areas was based on the SoftWare Engineering Body Of Knowledge (SWEBOK) knowledge areas and multiple discussions with dozens of SEEK area volunteers. Both the CCSE Steering Committee and the SEEK area volunteers felt strongly about emphasizing the academic discipline of software engineering. During the SEEK development process, the area chosen to represent the theoretical and scientific foundations of developing software products subsequently grew to the size of one half of the core. This prompted the Steering Committee to reevaluate whether the original goals of emphasizing the discipline were indeed being met. The resulting set of knowledge areas are believed to stress the fundamental principles, knowledge, and practices that underlie the software engineering discipline.

4.7 SE Education Knowledge Areas

In this section, we describe the ten knowledge areas that make up the SEEK: Computing Essentials (CMP), Mathematical & Engineering Fundamentals (FND), Professional Practice (PRF), Software Modeling & Analysis (MAA), Software Design (DES), Software Verification & Validation (VAV), Software Evolution (EVL), Software Process (PRO), Software Quality (QUA), and Software Management (MGT). The knowledge areas do not include material about continuous mathematics or the natural sciences; the needs in these areas will be discussed in other parts of the CCSE volume. For each knowledge area, there is a short paragraph description and then a table that delineates the units and topics for that area. Each area's topics are listed with one of three attributes: the Bloom's taxonomy level (what capability should a graduate possess concerning the topic), whether a topic is essential (or desirable or optional) to the core, and the recommended core contact hours for the unit.

Bloom's attributes are specified using one of the letters k, c, or a, which represent:

- 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).
- Comprehension (c) - understanding information and ability to grasp meaning of material presented. For example, translate knowledge to a new context, interpret facts, compare, contrast, order, group, infer causes, predict consequences, etc.
- Application (a) - ability to use learned material in new and concrete situations. For example, the use of information, methods, concepts, and theories to solve problems requiring the skills or knowledge presented.

A topic's relevance to the core is represented as follows:

- Essential (E) - the topic is part of the core.

- Desirable (D) - the topic is not part of the SEEK core, but it should be included in the core of a particular program if possible; otherwise, it should be considered as part of elective materials.
- Optional (O) - the topic should be considered as elective only.

4.8 Computing Essentials

Description

Computing essentials includes the computer science foundations that support the design and construction of software products. This area also includes knowledge about the transformation of a design into an implementation, the tools used during this process, and formal software construction methods.

Units and Topics

Reference		k,c,a	E,D,O	Hours	Related Topics
CMP	Computing Essentials			172	
CMP.cf	*Computer Science foundations*			140	
CMP.cf.1	Programming Fundamentals (CCCS PF1 to PF5) (control & data, typing, recursion)	a	E		
CMP.cf.2	Algorithms, Data Structures/Representation (static & dynamic) and Complexity (CCCS AL 1 to AL 5)	a	E		CMP.ct.1,CMP.fm.5,MAA.cc.1
CMP.cf.3	Problem solving techniques	a	E		CMP.cf.1
CMP.cf.4	Abstraction – use and support for (encapsulation, hierarchy, etc)	a	E		MAA.md.1
CMP.cf.5	Computer organization (parts of CCCS AR 1 to AR 5)	c	E		
CMP.cf.6	Basic concept of a system	c	E		MAA.rfd.7
CMP.cf.7	Basic user human factors (I/O, error messages, robustness)	c	E		DES.hci
CMP.cf.8	Basic developer human factors (comments, structure, readability)	c	E		CMP.cf.1
CMP.cf.9	Programming language basics (key concepts from CCCS PL1-PL6)	a	E		CMP.ct.3,CMP.ct.4
CMP.cf.10	Operating system basics (key concepts from CCCS OS1-OS5)	c	E		CMP.ct.10,CMP.ct.15
CMP.cf.11	Database basics	c	E		DES.con.2
CMP.cf.12	Network communication basics	c	E		
CMP.ct	*Construction technologies*			20	
CMP.ct.1	API design and use	a	E		DES.dd.4
CMP.ct.2	Code reuse and libraries	a	E		CMP.cf.1
CMP.ct.3	Object-oriented run-time issues (e.g. polymorphism, dynamic binding, etc.)	a	E		CMP.cf.1,9,DES.str.2
CMP.ct.4	Parameterization and generics	a	E		CMP.cf.1
CMP.ct.5	Assertions, design by contract, defensive programming	a	E		MAA.md.2
CMP.ct.6	Error handling, exception handling, and fault tolerance	a	E		DES.con.2,VAV.tst.2,VAV.tst.9
CMP.ct.7	State-based and table driven construction techniques	c	E		FND.mf.7,MAA.tm.2,CMP.cf.10
CMP.ct.8	Run-time configuration and internationalization	a	E		DES.hci.6
CMP.ct.9	Grammar-based input processing (parsing)	a	E		FND.mf.8
CMP.ct.10	Concurrency primitives (e.g. semaphores, monitors, etc.)	a	E		CMP.cf.10
CMP.ct.11	Middleware (components and containers)	c	E		DES.dd.3,5
CMP.ct.12	Construction methods for distributed software	a	E		CMP.cf.2

CMP.ct.13	Constructing heterogeneous (hardware and software) systems; hardware-software codesign	c	E		DES.ar.3
CMP.ct.14	Hot-spot analysis and performance tuning	k	E		FND.ef.4,DES.con.6,CMP.tl.4,VAV.fnd.4
CMP.ct.15	Platform standards (Posix etc.)		D		
CMP.ct.16	Test-first programming		D		VAV.tst.1
CMP.tl	*Construction tools*			4	DES.ste.1
CMP.tl.1	Development environments	a	E		
CMP.tl.2	GUI builders	c	E		DES.hci
CMP.tl.3	Unit testing tools	c	E		VAV.tst.1
CMP.tl.4	Application oriented languages (e.g. scripting, visual, domain-specific, markup, macros, etc.)	c	E		
CMP.tl.5	Profiling, performance analysis and slicing tools		D		CMP.ct.14
CMP.fm	*Formal construction methods*			8	DES.dd.9,MAA.af.6,EVO.ac.7
CMP.fm.1	Application of abstract machines (e.g. SDL, Paisley, etc.)	k	E		
CMP.fm.2	Application of specification languages and methods (e.g. ASM, B, CSP, VDM, Z)	a	E		MAA.md.3,MAA.rsd.3
CMP.fm.3	Automatic generation of code from a specification	k	E		
CMP.fm.4	Program derivation	c	E		
CMP.fm.5	Analysis of candidate implementations	c	E		MAA.cf.2
CMP.fm.6	Mapping of a specification to different implementations	k	E		
CMP.fm.7	Refinement	c	E		
CMP.fm.8	Proofs of correctness		D		FND.mf.3

4.9 Mathematical and Engineering Fundamentals

Description

The mathematical and engineering fundamentals of software engineering provide theoretical and scientific underpinnings for the construction of software products with desired attributes. These fundamentals support describing software engineering products in a precise manner. They provide the mathematical foundations to model and facilitate reasoning about these products and their interrelations, as well as form the basis for a predictable design process. A central theme is engineering design: a decision-making process of iterative nature, in which computing, mathematics, and engineering sciences are applied to deploy available resources efficiently to meet a stated objective.

Units and Topics

Reference		k,c,a	E,D,O	Hours	Related Topics
FND	**Mathematical and Engineering Fundamentals**			89	
FND.mf	*Mathematical foundations**			56	
FND.mf.1	Functions, Relations and Sets (CCCS DS1)	a	E		
FND.mf.2	Basic Logic (propositional and predicate) (CCCS DS2)	a	E		MAA.md.2,3
FND.mf.3	Proof Techniques (direct, contradiction, inductive) (CCCS DS3)	a	E		CMP.fm.8
FND.mf.4	Basic Counting (CCCS DS4)	a	E		
FND.mf.5	Graphs and Trees (CCCS DS5)	a	E		CMP.cf.2
FND.mf.6	Discrete Probability (CCCS DS6)	a	E		FND.ef.2
FND.mf.7	Finite State Machines, regular expressions	c	E		CMP.ct.7,MAA.t

					m.2
FND.mf.8	Grammars	c	E		CMP.ct.9
FND.mf.9	Numerical precision, accuracy and errors	c	E		
FND.mf.10	Number Theory		D		
FND.mf.11	Algebraic Structures		O		
FND.ef	*Engineering foundations for software*			23	
FND.ef.1	Empirical methods and experimental techniques (computer-related measuring techniques for CPU and memory usage)	c	E		VAV.fnd.4,VAV.hct.6
FND.ef.2	Statistical analysis (including simple hypothesis testing, estimating, regression, correlation etc.)	a	E		FND.mf.6
FND.ef.3	Measuring individual's performance (e.g. PSP)	k	E		PRO.con.5,PRO.imp.4
FND.ef.4	Systems development (e.g. security, safety, performance, effects of scaling, feature interaction, etc.)	k	E		MAA.af.4,DES.con.6,VAV.fnd.4,VAV.tst.9
FND.ef.5	Engineering design (e.g. formulation of problem, alternative solutions, feasibility, etc.)	c	E		FND.ec.3,MAA.af.1
FND.ef.6	Engineering science for other engineering disciplines (strength of materials, digital system principles, logic design, fundamentals of thermodynamics, etc.)		O		
FND.ec	*Engineering economics for software*			10	PRF.pr.6
FND.ec.1	Value considerations throughout the software lifecycle	k	E		
FND.ec.2	Generating system objectives (e.g. participatory design, stakeholder win-win, quality function deployment, prototyping, etc.)	c	E		PRF.psy.4,MAA.er.2
FND.ec.3	Evaluating cost-effective solutions (e.g. benefits realization, tradeoff analysis, cost analysis, return on investment, etc.)	c	E		DES.con.7,MAA.af.4,MGT.pp.4
FND.ec.4	Realizing system value (e.g. prioritization, risk resolution, controlling costs, etc.)	k	E		MAA.af.4,MGT.pp.6

* Topics 1-6 correspond to Computer Science curriculum guidelines for discrete structures 1-6

4.10 Professional Practice

Description

Professional Practice is concerned with the knowledge, skills, and attitudes that software engineers must possess to practice software engineering in a professional, responsible, and ethical manner. The study of professional practices includes the areas of technical communication, group dynamics and psychology, and social and professional responsibilities.

Units and Topics

Reference		k,c,a	E,D,O	Hours	Related Topics
PRF	**Professional Practice**			35	
PRF.psy	*Group dynamics / psychology*			5	
PRF.psy.1	Dynamics of working in teams/groups	a	E		
PRF.psy.2	Individual cognition (e.g. limits)	k	E		DES.hci.10
PRF.psy.3	Cognitive problem complexity	k	E		MAA.rfd.8
PRF.psy.4	Interacting with stakeholders	c	E		FND.ec.2
PRF.psy.5	Dealing with uncertainty and ambiguity	k	E		
PRF.com	*Communications skills (specific to SE)*			10	

PRF.com.1	Reading, understanding and summarizing reading (e.g. source code, documentation)	a	E		MAA.rsd.1
PRF.com.2	Writing (assignments, reports, evaluations, justifications, etc.)	a	E		
PRF.com.3	Team and group communication (both oral and written, email, etc.)	a	E		MGT.per
PRF.com.4	Presentation skills	a	E		
PRF.pr	*Professionalism*			20	
PRF.pr.1	Accreditation, certification, and licensing	k	E		
PRF.pr.2	Codes of ethics and professional conduct	c	E		
PRF.pr.3	Social, legal, historical, and professional issues and concerns	c	E		
PRF.pr.4	The nature of, and role of professional societies	k	E		
PRF.pr.5	The nature and role of software engineering standards	k	E		MAA.rsd.1,CMP.ct.14,PRO.imp.3,7,QUA.std
PRF.pr.6	The economic impact of software	c	E		FND.ec

4.11 Software Modeling and Analysis

Description

Modeling and analysis can be considered core concepts in any engineering discipline since they are essential to documenting and evaluating design decisions and alternatives. Modeling and analysis is first applied to the analysis, specification, and validation of requirements. Requirements represent the real world needs of users, customers and other stakeholders affected by the system and the capabilities and opportunities afforded by software and computing technologies. The construction of requirements includes an analysis of the feasibility of the desired system, elicitation and analysis of stakeholders' needs, the creation of a precise description of what the system should and should not do along with any constraints on its operation and implementation, and the validation of this description or specification by the stakeholders.

Units and Topics

Reference		k,c,a	E,D,O	Hours	Related Topics
MAA	**Software Modeling and Analysis**			53	
MAA.md	*Modeling foundations*			19	PRO.con.3,QUA.pro.1,QUA.pda.3
MAA.md.1	Modeling principles (e.g. decomposition, abstraction, generalization, projection/views, explicitness, use of formal approaches, etc.)	a	E		CMP.cf.4
MAA.md.2	Pre & post conditions, invariants	c	E		CMP.ct.5
MAA.md.3	Introduction to mathematical models and specification languages (Z, VDM, etc.)	c	E		MAA.rsd.3,CMP.fm.2
MAA.md.4	Properties of modeling languages	k	E		
MAA.md.5	Syntax vs. semantics (understanding model representations)	c	E		CMP.cf.9
MAA.md.6	Explicitness (make no assumptions, or state all assumptions)	k	E		
MAA.tm	*Types of models*			12	MAA.md
MAA.tm.1	Information modeling (e.g. entity-relationship modeling, class diagrams, etc.)	a	E		MAA.rsd.3,DES.dd.5
MAA.tm.2	Behavioral modeling (e.g. structured analysis, state diagrams, use case analysis, interaction diagrams, failure modes and	a	E		FND.mf.7,MAA.er.2,MAA.rsd.3,DE

	effects analysis, fault tree analysis etc.)				S.dd.5
MAA.tm.3	Structure modeling (e.g. architectural, etc.)	c	E		MAA.rfd.7
MAA.tm.4	Domain modeling (e.g. domain engineering approaches, etc.)	k	E		
MAA.tm.5	Functional modeling (e.g. component diagrams, etc.)	c	E		
MAA.tm.6	Enterprise modeling (e.g. business processes, organizations, goals, etc.)		D		
MAA.tm.7	Modeling embedded systems (e.g. real-time schedulability analysis, external interface analysis, etc.)		D		
MAA.tm.8	Requirements interaction analysis (e.g. feature interaction, house of quality, viewpoint analysis, etc.)		D		
MAA.tm.9	Analysis Patterns (e.g. problem frames, specification re-use, etc.)		D		
MAA.af	*Analysis fundamentals*			6	
MAA.af.1	Analyzing well-formedness (e.g. completeness, consistency, robustness, etc.)	a	E		
MAA.af.2	Analyzing correctness (e.g. static analysis, simulation, model checking, etc.)	a	E		
MAA.af.3	Analyzing quality (non-functional) requirements (e.g. safety, security, usability, performance, root cause analysis, etc.)	a	E		FND.ef.4,QUA.pda,DES.con.6,VAV.fnd.4,VAV.tst.9,VAV.hct,EVO.ac.4
MAA.af.4	Prioritization, trade-off analysis, risk analysis, and impact analysis	c	E		FND.ec.3,4,QUA.pda.4
MAA.af.5	Traceability	c	E		DES.ar.4,EVO.pro.2
MAA.af.6	Formal analysis	k	E		CMP.fm
MAA.rfd	*Requirements fundamentals*			3	
MAA.rfd.1	Definition of requirements (e.g. product, project, constraints, system boundary, external, internal, etc.)	c	E		
MAA.rfd.2	Requirements process	c	E		PRO.con.3
MAA.rfd.3	Layers/levels of requirements (e.g. needs, goals, user requirements, system requirements, software requirements, etc.)	c	E		MAA.rsd
MAA.rfd.4	Requirements characteristics (e.g. testable, non-ambiguous, consistent, correct, traceable, priority, etc.)	c	E		MAA.af.5
MAA.rfd.5	Managing changing requirements	c	E		MGT.ctl.1
MAA.rfd.6	Requirements management (e.g. consistency management, release planning, reuse, etc.)	k	E		CMP.ct.3
MAA.rfd.7	Interaction between requirements and architecture	k	E		MAA.tm.3,DES.ar.4,EVO.pro.2
MAA.rfd.8	Relationship of requirements to systems engineering, human-centered design, etc.		D		CMP.cf.6
MAA.rfd.9	Wicked problems (e.g. ill-structured problems; problems with many solutions; etc.)		D		PRF.psy.3
MAA.rfd.10	COTS as a constraint		D		
MAA.er	*Eliciting requirements*			4	
MAA.er.1	Elicitation Sources (e.g. stakeholders, domain experts, operational and organization environments, etc.)	c	E		PRF.psy.4
MAA.er.2	Elicitation Techniques (e.g. interviews, questionnaires/surveys, prototypes, use cases, observation, participatory techniques, etc.)	c	E		FND.ec.2,MAA.er.2
MAA.er.3	Advanced techniques (e.g. ethnographic, knowledge elicitation, etc.)		O		
MAA.rsd	*Requirements specification & documentation*			6	
MAA.rsd.1	Requirements documentation basics (e.g. types, audience, structure, quality, attributes, standards, etc.)	k	E		PRF.pr.5
MAA.rsd.2	Software requirements specification	a	E		

MAA.rsd.3	Specification languages (e.g. structured English, UML, formal languages such as Z, VDM, SCR, RSML, etc.)	k	E		MAA.md.3,CMP.fm.2
MAA.rv	*Requirements validation*			3	
MAA.rv.1	Reviews and inspection	a	E		MAA.rv.1,VAV.rev
MAA.rv.2	Prototyping to validate requirements (Summative prototyping)	k	E		
MAA.rv.3	Acceptance test design	c	E		VAV.tst.8
MAA.rv.4	Validating product quality attributes	c	E		QUA.cc.5
MAA.rv.5	Formal requirements analysis		D		MAA.af.1

4.12 Software Design

Description

Software design is concerned with issues, techniques, strategies, representations, and patterns used to determine how to implement a component or a system. The design will conform to functional requirements within the constraints imposed by other requirements such as resource, performance, reliability, and security. This area also includes specification of internal interfaces among software components, architectural design, data design, user interface design, design tools, and the evaluation of design.

Units and Topics

Reference		k,c,a	E,D,O	Hours	Related Topics
DES	**Software Design**			45	
DES.con	*Design concepts*			3	
DES.con.1	Definition of design	c	E		
DES.con.2	Fundamental design issues (e.g. persistent data, storage management, exceptions, etc.)	c	E		CMP.ct.6,VAV.tst.2,CMP.cf.11
DES.con.3	Context of design within multiple software development life cycles	k	E		
DES.con.4	Design principles (information hiding, cohesion and coupling)	a	E		
DES.con.5	Interactions between design and requirements	c	E		DES.ar.4
DES.con.6	Design for quality attributes (e.g. reliability, usability, performance, testability, fault tolerance, etc.)	k	E		FND.ef.4,MAA.tm.4,DES.ar.2,CMP.ct.14,VAV.fnd.4
DES.con.7	Design trade-offs	k	E		FND.ec.3,DES.ar.2,DES.ev
DES.con.8	Architectural styles, patterns, reuse	c	E		DES.ar,DES.dd.2,CMP.ct.3
DES.str	*Design strategies*			6	
DES.str.1	Function-oriented design	a c	E		
DES.str.2	Object-oriented design	c a	E		CMP.cf.9,DES.dd.5,CMP.ct.4
DES.str.3	Data-structure centered design		D		
DES.str.4	Aspect oriented design		O		
DES.ar	*Architectural design*			9	
DES.ar.1	Architectural styles (e.g. pipe-and-filter, layered, transaction-centered, peer-to-peer, publish-subscribe, event-based, client-server, etc.)	a	E		DES.con.8
DES.ar.2	Architectural trade-offs between various attributes	a	E		FND.ec.3

DES.ar.3	Hardware issues in software architecture	k	E		CMP.ct.13
DES.ar.4	Requirements traceability in architecture	k	E		MAA.af.5,DES.con.5,EVO.pro.2
DES.ar.5	Domain-specific architectures and product-lines	k	E		
DES.ar.6	Architectural notations (e.g. architectural structure viewpoints & representations, component diagrams, etc.)	c	E		MAA.tm
DES.hci	*Human computer interface design*			12	CMP.cf.7,VAV.hct,CMP.ct.2
DES.hci.1	General HCI design principles	a	E		
DES.hci.2	Use of modes, navigation	a	E		
DES.hci.3	Coding techniques and visual design (e.g. color, icons, fonts, etc.)	c	E		
DES.hci.4	Response time and feedback	a	E		
DES.hci.5	Design modalities (e.g. menu-driven, forms, question-answering, etc.)	a	E		
DES.hci.6	Localization and internationalization	c	E		CMP.ct.8
DES.hci.7	Human computer interface design methods	c	E		
DES.hci.8	Multi-media (e.g. I/O techniques, voice, natural language, web-page, sound, etc.)		D		
DES.hci.9	Metaphors and conceptual models		D		
DES.hci.10	Psychology of HCI		D		PRF.psy.2
DES.dd	*Detailed design*			12	
DES.dd.1	One selected design method (e.g. SSA/SD, JSD, OOD, etc.)	a	E		
DES.dd.2	Design patterns	a	E		DES.con.8
DES.dd.3	Component design	a	E		CMP.ct.11
DES.dd.4	Component and system interface design	a	E		CMP.ct.2
DES.dd.5	Design notations (e.g. class and object diagrams, UML, state diagrams, etc.)	c	E		MAA.tm
DES.ste	*Design support tools and evaluation*			3	
DES.ste.1	Design support tools (e.g. architectural, static analysis, dynamic evaluation, etc.)	a	E		CMP.ct
DES.ste.2	Measures of design attributes (e.g. coupling, cohesion, information-hiding, separation of concerns, etc.)	k	E		
DES.ste.3	Design metrics (e.g. architectural factors, interpretation, metric sets in common use, etc.)	a	E		
DES.ste.4	Formal design analysis		O		MAA.af.2

4.13 Software Verification and Validation

Description

Software verification and validation uses both static and dynamic techniques of system checking to ensure that the resulting program satisfies its specification and that the program as implemented meets the expectations of the stakeholders. Static techniques are concerned with the analysis and checking of system representations throughout all stages of the software life cycle while dynamic techniques involve only the implemented system.

Units and Topics

Reference		k,c,a	E,D,O	Hours	Related Topics
VAV	**Software Verification and Validation**			42	
VAV.fnd	*V&V terminology and foundations*			5	

VAV.fnd.1	Objectives and constraints of V&V	k	E		
VAV.fnd.2	Planning the V&V effort	k	E		
VAV.fnd.3	Documenting V&V strategy, including tests and other artifacts	a	E		
VAV.fnd.4	Metrics & Measurement (e.g. reliability, usability, performance, etc.)	k	E		FND.ef.4,MAA.af.2,DES.con.6,CMP.ct.14,PRO.con.4
VAV.fnd.5	V&V involvement at different points in the lifecycle	k	E		
VAV.rev	*Reviews*			6	MAA.rv.1
VAV.rev.1	Desk checking	a	E		
VAV.rev.2	Walkthroughs	a	E		
VAV.rev.3	Inspections	a	E		VAV.hct.2,3
VAV.tst	*Testing*			21	MAA.rfd.4,DES.con.6,CMP.ct.15
VAV.tst.1	Unit testing	a	E		CMP.ct.15,CMP.ct.3
VAV.tst.2	Exception handling (writing test cases to trigger exception handling; designing good handling)	a	E		DES.con.2,CMP.ct.6
VAV.tst.3	Coverage analysis (e.g. statement, branch, basis path, multi--condition, dataflow, etc.)	a	E		
VAV.tst.4	Black-box functional testing techniques	a	E		
VAV.tst.5	Integration Testing	c	E		
VAV.tst.6	Developing test cases based on use cases and/or customer stories	a	E		MAA.tm.2
VAV.tst.7	Operational profile-based testing	k	E		
VAV.tst.8	System and acceptance testing	a	E		MAA.rv.4
VAV.tst.9	Testing across quality attributes (e.g. usability, security, compatibility, accessibility, etc.)	a	E		MAA.af.3,MAA.rv.6,VAV.hct,QUA.cc.5
VAV.tst.10	Regression Testing	c	E		
VAV.tst.11	Testing tools	a	E		CMP.ct.3
VAV.tst.12	Deployment process		D		
VAV.hct	*Human computer user interface testing and evaluation*			6	DES.hci,VAV.tst.9
VAV.hct.1	The variety of aspects of usefulness and usability	k	E		MAA.af.3
VAV.hct.2	Heuristic evaluation	a	E		VAV.rev.3
VAV.hct.3	Cognitive walkthroughs	c	E		VAV.rev.3
VAV.hct.4	User testing approaches (observation sessions etc.)	a	E		
VAV.hct.5	Web usability; testing techniques for web sites	c	E		
VAV.hct.6	Formal experiments to test hypotheses about specific HCI controls		D		FND.ef.1
VAV.par	*Problem analysis and reporting*			4	
VAV.par.1	Analyzing failure reports	c	E		
VAV.par.2	Debugging/fault isolation techniques	a	E		
VAV.par.3	Defect analysis	k	E		
VAV.par.4	Problem tracking	c	E		

4.14 Software Evolution

Description

Software evolution is the result of the ongoing need to support the stakeholders' mission in the face of changing assumptions, problems, requirements, architectures and technologies. It is intrinsic to all real world software systems. Support for evolution requires numerous activities both before and after each of a succession of versions or upgrades (releases) that constitute the evolving system. Evolution is a broad concept that expands upon the traditional notion of software maintenance.

Units and Topics

Reference		k,c,a	E,D,O	Hours	Related Topics
EVO	Software Evolution			10	
EVO.pro	*Evolution processes*			6	
EVO.pro.1	Basic concepts of evolution and maintenance	k	E		
EVO.pro.2	Relationship between evolving entities (e.g. assumptions, requirements, architecture, design, code, etc.)	k	E		MAA.af.4,DES.ar.4
EVO.pro.3	Models of software evolution (e.g. theories, laws, etc.)	k	E		
EVO.pro.4	Cost models of evolution		D		FND.ec.3
EVO.pro.5	Planning for evolution (e.g. outsourcing, in-house, etc.)		D		MGT.pp
EVO.ac	Evolution activities			4	VAV.par.4,MGT.cm
EVO.ac.1	Working with legacy systems (e.g. use of wrappers, etc.)	k	E		
EVO.ac.2	Program comprehension and reverse engineering	k	E		
EVO.ac.3	System and process re-engineering (technical and business)	k	E		
EVO.ac.4	Impact analysis	k	E		
EVO.ac.5	Migration (technical and business)	k	E		
EVO.ac.6	Refactoring	k	E		
EVO.ac.7	Program transformation		D		
EVO.ac.8	Data reverse engineering		D		

4.15 Software Process

Description

Software process is concerned with knowledge about the description of commonly used software life-cycle process models and the contents of institutional process standards; definition, implementation, measurement, management, change and improvement of software processes; and use of a defined process to perform the technical and managerial activities needed for software development and maintenance.

Units and Topics

Reference		k,c,a	E,D,O	Hours	Related Topics
PRO	Software Process			13	
PRO.con	*Process concepts*			3	
PRO.con.1	Themes and terminology	k	E		
PRO.con.2	Software engineering process infrastructure (e.g. personnel, tools, training, etc.)	k	E		
PRO.con.3	Modeling and specification of software processes	c	E		MAA.rfd.2
PRO.con.4	Measurement and analysis of software processes	c	E		MGT.ctl.3

PRO.con.5	Software engineering process improvement (individual, team)	c	E		FND.ef.3,PRO.imp.4,5
PRO.con.6	Quality analysis and control (e.g. defect prevention, review processes, quality metrics, root cause analysis, etc.)	c	E		MAA.rv.1,VAV.rev,QUA.pda.4
PRO.con.7	Analysis and modeling of software process models		D		
PRO.imp	*Process implementation*			10	
PRO.imp.1	Levels of process definition (e.g. organization, project, team, individual, etc.)	k	E		
PRO.imp.2	Life cycle models (agile, heavyweight, waterfall, spiral, etc.)	c	E		DES.con.3,VAV.fnd.5
PRO.imp.3	Life cycle process models and standards (e.g., IEEE, ISO, etc.)	c	E		PRF.pr.5,QUA.pro.2
PRO.imp.4	Individual software process (model, definition, measurement, analysis, improvement)	a	E		PRO.con.5
PRO.imp.5	Team software process (model, definition, organization, measurement, analysis, improvement)	a	E		PRO.con.5
PRO.imp.6	Process tailoring	k	E		
PRO.imp.7	ISO/IEEE Standard 12207: requirements of processes	k	E		PRF.pr.5

4.16 Software Quality

Description

Software quality is a pervasive concept that affects, and is affected by all aspects of software development, support, revision, and maintenance. It encompasses the quality of work products developed and/or modified (both intermediate and deliverable work products) and the quality of the work processes used to develop and/or modify the work products. Quality work product attributes include usability, reliability, safety, security, maintainability, flexibility, efficiency, performance and availability.

Units and Topics

Reference		k,c,a	E,D,O	Hours	Related Topics
QUA	**Software Quality**			16	
QUA.cc	*Software quality concepts and culture*			2	
QUA.cc.1	Definitions of quality	k	E		
QUA.cc.2	Society's concern for quality	k	E		
QUA.cc.3	The costs and impacts of bad quality	k	E		
QUA.cc.4	A cost of quality model	c	E		MGT.pp.4
QUA.cc.5	Quality attributes for software (e.g. dependability, usability, etc.)	k	E		MAA.rva.5,VAV.tst.9,QUA.pda.5
QUA.cc.6	The dimensions of quality engineering	k	E		
QUA.cc.7	Roles of people, processes, methods, tools, and technology	k	E		
QUA.std	*Software quality standards*			2	PRF.pr.5
QUA.std.1	The ISO 9000 series	k	E		
QUA.std.2	ISO/IEEE Standard 12207: the "umbrella" standard	k	E		
QUA.std.3	Organizational implementation of standards	k	E		
QUA.std.4	IEEE software quality-related standards		D		
QUA.pro	*Software quality processes*			4	
QUA.pro.1	Software quality models and metrics	c	E		VAV.fnd.4,QUA.pda.5

QUA.pro.2	Quality-related aspects of software process models	k	E		PRO.imp.3
QUA.pro.3	Introduction/overview of ISO 15504 and the SEI CMMs	k	E		PRF.pr.5
QUA.pro.4	Quality-related process areas of ISO 15504	k	E		PRF.pr.5
QUA.pro.5	Quality-related process areas of the SW-CMM and the CMMIs	k	E		
QUA.pro.6	The Baldridge Award criteria for software engineering		O		
QUA.pro.7	Quality aspects of other process models		O		
QUA.pca	*Process assurance*			4	
QUA.pca.1	The nature of process assurance	k	E		
QUA.pca.2	Quality planning	a	E		MGT.pp
QUA.pca.3	Organizing and reporting for process assurance	a	E		
QUA.pda.4	Techniques of process assurance	c	E		
QUA.pda	Product assurance			4	
QUA.pda.1	*The nature of product assurance*	k	E		
QUA.pda.2	*Distinctions between assurance and V&V*	k	E		VAV
QUA.pda.3	*Quality product models*	k	E		
QUA.pda.4	*Root cause analysis and defect prevention*	c	E		PRO.con.6
QUA.pda.5	*Quality product metrics and measurement*	c	E		VAV.fnd.4,QUA.cc.5,QUA.pro.1
QUA.pda.6	*Assessment of product quality attributes (e.g. useability, reliability, availability, etc.)*	c	E		

4.17 Software Management

Description

Software management is concerned with knowledge about the planning, organization, and monitoring of all software life cycle phases. Management is critical to ensure that software development projects are appropriate to an organization, work in different organizational units is coordinated, software versions and configurations are maintained, resources are available when necessary, project work is divided appropriately, communication is facilitated, and progress is accurately charted.

Units and Topics

Reference		k,c,a	E,D,O	Hours	Related Topics
MGT	**Software Management**			19	
MGT.con	*Management concepts*			2	
MGT.con.1	General project management	k	E		
MGT.con.2	Classic management models	k	E		
MGT.con.3	Project management roles	k	E		
MGT.con.4	Enterprise/Organizational management structure	k	E		
MGT.con.5	Software management types (e.g. acquisition, project, development, maintenance, risk, etc.)	k	E		FND.ec.4,MGT.pp.6,EVO
MGT.pp	*Project planning*			6	VAV.fnd.2,QUA.pca.2
MGT.pp.1	Evaluation and planning	c	E		
MGT.pp.2	Work breakdown structure	a	E		
MGT.pp.3	Task scheduling	a	E		
MGT.pp.4	Effort estimation	a	E		FND.ec.3,QUA.cc.4

MGT.pp.5	Resource allocation	c	E		
MGT.pp.6	Risk management	a	E		FND.ec.4
MGT.per	*Project personnel and organization*			2	PRF.com.3
MGT.per.1	Organizational structures, positions, responsibilities, and authority	k	E		
MGT.per.2	Formal/informal communication	k	E		
MGT.per.3	Project staffing	k	E		
MGT.per.4	Personnel training, career development, and evaluation	k	E		
MGT.per.5	Meeting management	a	E		
MGT.per.6	Building and motivating teams	a	E		
MGT.per.7	Conflict resolution	a	E		
MGT.ctl	*Project control*			4	
MGT.ctl.1	Change control	k	E		MAA.rfd.5,MGT.cm.1,2
MGT.ctl.2	Monitoring and reporting	c	E		
MGT.ctl.3	Measurement and analysis of results	c	E		PRO.con.4
MGT.ctl.4	Correction and recovery	k	E		
MGT.ctl.5	Reward and discipline		O		
MGT.ctl.6	Standards of performance		O		
MGT.cm	*Software configuration management*			5	
MGT.cm.1	Revision control	a	E		MGT.ctl.1
MGT.cm.2	Release management	c	E		MGT.ctl.1
MGT.cm.3	Tool support	c	E		
MGT.cm.4	Builds	c	E		
MGT.cm.5	Software configuration management processes	k	E		
MGT.cm.6	Maintenance issues	k	E		EVO.ac
MGT.cm.7	Distribution and backup		D		

4.18 Systems and Application Specialties

As part of an undergraduate software engineering education, students should specialize in one or more areas. Within their specialty, students should learn material well beyond the core material specified above. They may either specialize in one or more of the ten knowledge areas listed above, or they may specialize in one or more of the application areas listed below. For each application area, students should obtain breadth in the related domain knowledge while they are obtaining a depth of knowledge about the design of a particular system. Students should also learn about the characteristics of typical products in these areas and how these characteristics influence a system's design and construction. Each application specialty listed below is elaborated with a list of related topics that are needed to support the application.

This list of application areas is not intended to be exhaustive but is designed to give guidance to those developing specialty curricula.

Specialties and Their Related Topics

Reference	
SAS	**System and Application Specialties**
SAS.net	*Network-centric systems*

SAS.net.1	Knowledge and skills in web-based technology
SAS.net.2	Depth in networking
SAS.net.3	Depth in security
SAS.inf	Information systems and data processing
SAS.inf.1	Depth in databases
SAS.inf.2	Depth in business administration
SAS.inf.3	Data warehousing
SAS.fin	*Financial and e-commerce systems*
SAS.fin.1	Accounting
SAS.fin.2	Finance
SAS.fin.3	Depth in security
SAS.sur	Fault tolerant and survivable systems
SAS.sur.1	Knowledge and skills with heterogeneous, distributed systems
SAS.sur.2	Depth in security
SAS.sur.3	Failure analysis and recovery
SAS.sur.4	Intrusion detection
SAS.sec	Highly secure systems
SAS.sec.1	Business issues related to security
SAS.sec.2	Security weaknesses and risks
SAS.sec.3	Cryptography, cryptanalysis, steganography, etc.
SAS.sec.4	Depth in networks
SAS.sfy	Safety critical systems
SAS.sfy.1	Depth in formal methods, proofs of correctness, etc.
SAS.sfy.2	Knowledge of control systems
SAS.emb	Embedded and real-time systems
SAS.emb.1	Hardware for embedded systems
SAS.emb.2	Language and tools for development
SAS.emb.3	Depth in timing issues
SAS.emb.3	Hardware verification
SAS.bio	Biomedical systems
SAS.bio.1	Biology and related sciences
SAS.bio.2	Related safety critical systems knowledge
SAS.sci	Scientific systems
SAS.sci.1	Depth in related science
SAS.sci.2	Depth in statistics
SAS.sci.3	Visualization and graphics
SAS.tel	Telecommunications systems
SAS.tel.1	Depth in signals, information theory, etc.
SAS.tel.2	Telephony and telecommunications protocols
SAS.av	Avionics and vehicular systems
SAS.av.1	Mechanical engineering concepts
SAS.av.2	Related safety critical systems knowledge
SAS.av.3	Related embedded and real-time systems knowledge

SAS.ind	Industrial process control systems
SAS.ind.1	Control systems
SAS.ind.2	Industrial engineering and other relevant areas of engineering
SAS.ind.3	Related embedded and real-time systems knowledge
SAS.mm	Multimedia, game and entertainment systems
SAS.mm.1	Visualization, haptics, and graphics
SAS.mm.2	Depth in human computer interface design
SAS.mm.3	Depth in networks
SAS.mob	Systems for small and mobile platforms
SAS.mob.1	Wireless technology
SAS.mob.2	Depth in human computer interfaces for small and mobile platforms
SAS.mob.3	Related embedded and real-time systems knowledge
SAS.mob.4	Related telecommunications systems knowledge
SAS.ab	Agent-based systems
SAS.ab.1	Machine learning
SAS.ab.2	Fuzzy logic
SAS.ab.3	Knowledge engineering

Chapter 5: Guidelines for SE Curriculum Design and Delivery

Chapter 4 of this document presented SEEK, the knowledge that software engineering graduates need to be taught. However, *how* the SEEK topics should be taught may be as important as *what* is taught. This chapter presents a set of guidelines that should be considered by those developing an undergraduate SE curriculum, and by those teaching individual SE courses.

5.1 Guideline regarding those developing and teaching the curriculum

Curriculum Guideline 1: Curriculum designers and instructors must have sufficient knowledge and experience such that they understand clearly the character of software engineering.

Software engineering can mean different things to different people. However, those who have experienced a wide variety of software projects, and read a wide variety of software engineering literature, tend to have views of software engineering that converge towards the consensus presented in this document.

Curriculum designers and instructors should therefore:

- Have deep, and broad software engineering knowledge in most areas of SEEK and SWEBOK.
- Have, or work towards obtaining, real world experience in software engineering. Academics in research careers could obtain this by performing research in an industrial setting where they work closely with software engineers.
- Become recognized publicly as knowledgeable in software engineering, either by having a track record of publication, or by being certified in some way (such as the IEEE CSDP certification, or other such designations offered by a professional engineering society).

Failure to adhere to this principle will open a program or course to certain risks:

- A program or course might be biased excessively to one kind of software or class of methods, thus nor giving students a broad enough exposure to the field, or an inaccurate perception of the field. For example, instructors who have only experienced real-time or data-processing systems are at risk of focusing their programs excessively towards such systems. While it is not bad to have programs that are specialized towards specific types of software engineering, such specializations must be explicitly acknowledged in the course titles of more advanced courses. At the introductory levels, the material taught should be broadly applicable, and example problems should be derived from many types of applications and approaches.
- Faculty developing software engineering programs who have a primarily theoretical computer science background might not adequately convey to students the engineering-nature of software engineering
- Faculty from related branches of engineering might deliver a software engineering program or course without a full appreciation of the computer science fundamentals that underlie so

much of what software engineers do, as well as of the wide range of domains beyond engineering to which software engineering can be applied.

- Faculty who have not experienced the development of large systems, might not appreciate the importance of process, quality, evolution and management (which are knowledge areas of the SEEK).
- Faculty who have made a research career out of pushing the frontiers of software development, might not appreciate that students need first to be taught what they can use in practice and need to understand the motivation behind what they are taught.

5.2 Guidelines for constructing the curriculum

Curriculum Guideline 2: Curriculum designers and instructors must think in terms of outcomes

Both entire programs and individual courses should be designed starting with outcomes. Furthermore, as courses are taught these outcomes should be regularly kept in mind. Thinking in terms of outcomes helps ensure that the material included in the curriculum is relevant and is taught in an appropriate manner and an appropriate level of depth.

The CCSE Graduate Outcomes (See … - to be added) should be used as a basis for designing and assessing software engineering curricula in general. These can be further specialized for the design of individual courses. In addition, particular institutions may develop more focused or detailed outcomes – e.g. abilities in certain applications areas, or deeper abilities in certain SEEK knowledge areas.

Curriculum Guideline 3: Curriculum designers must strike an appropriate balance between coverage of material, and flexibility to allow for innovation

In deciding what should be taught in a course, there is a temptation to fill the course up with a list of material that must be covered. For example, in the case of a course consisting of 40 hours of lectures, the temptation is to allocate all 40 hours to particular SEEK essential topics. Unfortunately, doing so would result in a curriculum that left no space for desirable and optional topics (except in elective courses), and would result in an inability to innovate on the part of instructors.

This guideline applies most strongly in more advanced courses.

Curriculum Guideline 4: Many SE concepts, principles and issues should be taught as recurring themes throughout the curriculum to help students develop a software engineering mindset.

Material defined in many SEEK units should be taught in a manner that is distributed through many courses in the curriculum. Generally, early courses should introduce the material, with subsequent courses reinforcing and expanding upon the material. In most cases, there should also be courses or parts of courses that treat the material in depth.

In addition to ethics and tool use, which will be highlighted specifically in other guidelines, the following are types of material that should be presented, at least in part, as recurring themes:

- Measurement, quantification and formal or mathematical approaches
- Modeling, representation and abstraction
- Human factors
- Much of the material in the Process, Quality, Evolution and Management knowledge areas.

Curriculum Guideline 5: Certain types of material that require maturity should be taught later, while other material should be taught earlier to facilitate gaining that maturity

If taught too early, many topics from SEEK's Process, Quality, Evolution, and Management knowledge areas are likely to be poorly understood and appreciated by students. This should be taken into account when designing the sequence with which material is to be taught and how real-world experiences are introduced to the students. It is suggested that introductory material on these topics can be taught in early years, but that the bulk of the material be left to the latter half of the curriculum.

On the other hand, students also need very practical material to be taught early so they can begin to gain maturity by participating in real-world development experiences (in the work force or in student projects). Examples of topics whose teaching should start early include programming, human factors, aspects of requirements and design, as well as verification and validation. This does not mean to imply that programming has to be taught first, as in a traditional CS1 course, but that at least a reasonable amount should be taught in a student's first year.

Curriculum Guideline 6: Students must learn some application domain or domains outside of software engineering.

Almost all software engineering activity will involve solving problems for customers in domains outside software engineering. Therefore, somewhere in their curriculum, students should be able to study one or more outside domains in reasonable depth.

Studying such material will not only give the student direct domain knowledge they can apply to software engineering problems, but will also teach them the language and thought processes of the domain, enabling more in-depth study later on.

By 'in reasonable depth' we mean one or more courses that are at more than the introductory level (at least heavy second year courses and beyond). The choices of domain or domains is up to the institution or can be left to the student. They can include other branches of engineering or the natural sciences; they can also include social sciences, business and the humanities. No one domain should be considered 'more important' to software engineering programs than another.

The study of certain domains will necessitate additional supporting courses, such as particular areas of mathematics and computer science as well as deeper areas of software engineering. The reader should consult the Systems and Application Specialties area at the end of SEEK (Chapter 4) to see recommendations for such supporting courses.

This guideline does not preclude the possibility of designing courses or programs that deeply integrate the teaching of domain knowledge with the teaching of software engineering. In fact, such an approach would be innovative and commendable. For example, an institution could have courses called 'Telecommunications Software Engineering', 'Aerospace Software Engineering', ‘Information Systems Software Engineering’, or ‘Software Engineering of Sound and Music Systems’. However, in such cases great care must be taken to ensure that the depth is not sacrificed in either SE or the domain. The risk is that the instructor, the instructional material, or the presentation may not have adequate depth in one or the other area.

5.3 Attributes and attitudes that should pervade the curriculum and its delivery

Curriculum Guideline 7: Software engineering must be taught in ways that emphasize its engineering nature

In order for software engineering to take its place alongside older branches of engineering, educators must develop an appreciation for those aspects of software engineering that it shares in common with other branches. Engineering has been evolving for millennia, and a great deal of general wisdom has been built up. Software engineering educators must embrace that wisdom, at the same time realizing that some parts of it need to be adapted to the software engineering context.

Software engineering programs and courses must therefore embrace the characteristics of engineering that are presented in Chapter 3.

In addition, software engineering students must develop a sense of the engineering ethos, and an understanding of the responsibilities of being an engineer. This can only be achieved by appropriate attitudes on the part of all faculty and administrators.

Curriculum Guideline 8: Students should be trained to exercise critical judgment

Making judgments among competing solutions is a key part of what it means to be an engineer. Curriculum design and delivery should therefore help students build the knowledge, analysis skills and methods they need to make sound judgments. Of particular importance is a willingness to think critically.

Curriculum Guideline 9: Students should be instilled with the ability and eagerness to learn by themselves

Since so much of what is learned will change over a student’s professional career, and since only a small fraction of what could be learned will be taught and learned at university, it is of paramount importance that students develop the habit of continually expanding their knowledge.

Curriculum Guideline 10: Software engineering must be taught as a problem-solving discipline.

The ultimate goal of all software projects is solving customers’ problems; it is important to recognize this when designing programs and courses. Such recognition focuses the learner on the rationale for what he or she is learning, deepens the understanding of the knowledge learned, and helps ensure that the material taught is relevant.

There are a variety of classes of problems, all of which are important: Some, such as analysis, design, and testing problems, are product-oriented and are aimed directly at solving the customers' problem. Others, such as process improvement, are meta-problems – whose solution will facilitate the product-oriented, problem-solving process. Still others, such as ethical problems, transcend the above two categories.

Problem solving must be learned through practice, and must be taught through examples. Having a teacher show a solution on the screen can go part of the way, but is never sufficient. Students therefore must be given a significant number of assignments. The need for lecturing can be reduced if the example problems are well described in textbooks or on-line material.

Curriculum Guideline 11: The underlying and enduring *principles* of software engineering should be emphasized, rather than *details* of the latest or specific tools.

SEEK lists many topics that can be taught using a variety of different computer hardware, software applications, technologies, and processes (which we will refer to collectively as tools). In a good curriculum, it is the enduring knowledge in the SEEK topics that must be emphasized, not the details of the tools. The SEEK topics are supposed to remain valid for many years; the knowledge and experience derived from their learning should still be applicable 10 or 20 years later. Particular tools, on the other hand, will rapidly change. It is a mistake, for example, to focus excessively on how to use a particular vendor's piece of software, on the detailed steps of a methodology, or on the syntax of a programming language.

Applying this guideline to programming languages requires understanding that the line between what is enduring and what is temporary can be somewhat hard to pinpoint, and can be a moving target. It is clear that software engineers should learn several programming languages in detail, as well as other types of languages such as visual and formal specification languages. This guideline must be therefore be interpreted as saying that when learning such languages, students must learn much more than just surface syntax, and, having learned the languages, should be able to learn whatever new languages appear with little difficulty.

Applying this guideline to processes (also known as 'methods' or 'methodologies') is similar to applying it to languages. Students ought not to have to memorize long lists of steps, but should instead learn the underlying wisdom behind the steps such that they can choose whatever methodologies appear in the future, and can creatively adapt and mix processes.

Applying this guideline to technologies (both hardware and software) means not having to memorize in detail an API, user interface or instruction set just for the sake of memorizing it. Instead, students should develop the skill of looking up details in a reference manual whenever needed, so they can concentrate on more important matters.

Curriculum Guideline 12: The curriculum must be taught so that students gain experience using appropriate and up-to-date tools, even though tool details are not the focus of the learning.

To perform software engineering efficiently and effectively requires choosing and using the most appropriate computer hardware, software tools, technologies and processes (again, collectively referred to as tools). Students must therefore be habituated to choosing and using tools, so they go into the workforce with this habit – a habit that is often hard to pick up in the workforce, where the pressure to deliver results can often cause people to hesitate to learn new tools.

Appropriateness of tools must be carefully considered. A tool that is too complex, too unreliable, too expensive, too hard to learn given the available time and resources, or provides too little benefit, is inappropriate, whether in the educational context or in the work context. Many software engineering tools have failed because they have failed this criterion. Tools should be selected that support the process of learning principles.

Tools used in curricula must be reasonably up-to-date for several reasons: a) so that students can take the tools into the workplace as 'ambassadors'– performing a form of technology transfer; b) so that students can take advantage of the tool skills they have learned; c) so that students and employers will not feel the education is out of-date, even if up-to-date principles are being taught. Conversely, older tools can sometimes be simpler, and therefore more appropriate.

This guideline may seem in conflict with Curriculum Guideline 11, but that conflict is illusory. The key to avoiding the conflict is recognizing that teaching using tools does not mean that the object of the teaching is the tools themselves. Learning to use tools should be a secondary activity performed in laboratory or tutorial sessions, or by the student on his or her own. Students should realize that the tools are only aids, and they should learn not to fear learning new tools.

Curriculum Guideline 13: Material taught in a software engineering program should, where possible, be grounded in sound research and mathematical theory, or widely-accepted best practice

There must be evidence that whatever is taught is indeed true and useful. This evidence can take the form of validated scientific or mathematical theory (such as in many areas of computer science), or else widely-used and generally accepted practice.

It is important, however, not to be overly dogmatic about the application of theory: It may not always be appropriate. For example, formalizing of a specification or design so as to be able to apply mathematical approaches can be inefficient and reduce agility in many situations. In other circumstances, however, it may be essential.

In situations where material taught is based on generally accepted practice that has not yet been scientifically validated, the fact that the material is still open to question should be made clear.

This guideline complements Curriculum Guideline 11. Whereas curriculum Guideline 11 says focus on fundamental software engineering principles, Curriculum Guideline 13 says that what is taught should be well-founded.

Curriculum Guideline 14: The curriculum should have a significant real-world basis

Incorporating real-world elements into the curriculum is necessary to enable effective learning of software engineering skills and concepts   A program should be set up to incorporate at least some of the following:

- **Case studies**: Exposure to real systems and project case studies, taught to critique these as well as to reuse the best parts of them.
- **Project-based classes**: Some courses should be set up to mimic typical projects in industry. These should include group-work, presentations, formal reviews, quality assurance, etc. It would be beneficial if such a course were to include a real-world customer or customers. Projects should be interdisciplinary when possible.

- **Capstone Course(s)**: Students need a significant project, preferably spanning their entire last year, in order to practice the knowledge and skills they have learned. Unlike project-based classes, the capstone project is managed by the students and may solve a problem of the students' choice. It should normally be done in a group. Discussion of a capstone course in the curriculum can be found in Section 7.3.3.
- **Practical Exercises:** Students should be given practical exercises so they can develop skills in current practices and processes.
- **Student work experience**: Where possible, students should have some form of industrial work experience as a part of their program. This could take the form of one or more internships, co-op work terms, or sandwich work terms (the terminology used here is clearly country-dependent). It is desirable, although not always possible, to make work experience compulsory. If opportunities for work experience are difficult to provide, then simulation of work experience must be achieved in courses.

Curriculum Guideline 15: Ethical concerns, and the notion of what it means to be a professional, should be raised frequently.

One of the key reasons for the existence of a defined profession is to ensure that its members follow ethical principles and professional principles. By taking opportunities to discuss these issues throughout the curriculum, they will be come deeply entrenched. See Section 3.3 for further discussion of professionalism.

5.4 General strategies for software engineering pedagogy

Curriculum Guideline 16: In order to ensure students embrace certain important ideas, care must be taken to motivate students by using interesting, concrete and convincing examples.

It may be only through bitter experience that software engineers learn certain concepts and techniques considered central to the discipline. In some cases educators have not appreciated the value of these concepts and therefore have not taught them. But in many cases, educators have taught such concepts at a superficial level, but have failed to convince students as to their importance or veracity. In fact, educators sometimes encounter skepticism or outright derision when trying to teach certain ideas.

In these cases, there is a need to put considerable attention into motivating students to accept the ideas, by using interesting, concrete and revealing examples. The examples should be of sufficient size and complexity so as to demonstrate that using the material being taught has obvious benefits, and that failure to use the material would lead to undesirable consequences.

The following are examples of areas where motivation is particularly needed:

- Mathematical foundations: Logic and discrete mathematics should be taught in the context of its *application* to software engineering or computer science problems. If derivations and proofs are to be presented, these should preferably be taught following motivation of why the result is important. Statistics and empirical methods should likewise be taught in an applied, rather than abstract, manner.

- Process and quality: Students should be exposed to the consequences of poor processes and bad quality. They should also be exposed to good processes and quality so they can experience for themselves the effect of improvements.
- Human factors and usability: Students will often not appreciate the need for attention to these areas unless they actually experience usability difficulties, or watch users having difficulty using software.

Curriculum Guideline 17: Software engineering education in the 21st century needs to move beyond the lecture format: It is therefore important to encourage consideration of a variety of teaching and learning approaches.

The most common approach to teaching software engineering material is the use of lectures, supplemented by laboratory sessions, tutorials, etc. However, there are many who believe that alternative approaches can help students learn more effectively. Some of the approaches that should be considered to supplement or even largely replace the lecture format include:

- Problem-based learning: This has been found to be particularly useful in other professional disciplines, and is now used to teach engineering in some institutions. See Curriculum Guideline 10 for a discussion of the problem-solving nature of the discipline.
- Just-in-time learning: Teaching fundamental material immediately before teaching the application of that material. For example, teaching aspects of mathematics the day before they are applied in a software engineering context. There is evidence that this helps students retain the fundamental material, although it can be difficult to accomplish since faculty must co-ordinate across courses.
- Self-study materials that students work through on their own schedule.

Curriculum Guideline 18: Important efficiencies and synergies can be achieved by designing curricula so that several types of knowledge are learned at the same time

Many people browsing through SEEK have commented that there is a very large amount of material to be taught, or contrarily, that many topics are assigned a rather small number of hours. However, if careful attention is paid to the curriculum, many SEEK topics can be taught concurrently; in fact two topics listed as requiring x and y hours respectively may be taught together in less than x+y hours.

The following are some of the many situations where such synergistic teaching and learning may be applied:

- Modeling, languages and notations: Considerable depth in languages such as UML can be achieved by merely using the notation when teaching other concepts. The same applies to formal methods and programming. Clearly there will need to be some time set aside to teach the basics of a language or modeling technique *per se*, but both broad and deep knowledge can be learned as students study a wide range of other topics.
- Process, quality and management: Students can be instructed to follow certain processes as they are working on exercises or projects whose explicit objective is to learn other concepts. In these circumstances it would be desirable for students to have had some introduction to process so they know why they are being asked to follow a process. Also, it might be desirable to follow the exercise or project with a discussion of the usefulness of applying the

particular process. The depth of learning of the process is likely to be considerable, with relatively little time being taken away from the other material being taught.

- Mathematics: Students might deepen and expand their understanding of statistics while analyzing some data resulting from studies of reliability or performance. Opportunities to deepen understanding of logic and other branches of discrete mathematics also abound.

Teaching multiple concepts at the same time in this manner can, in fact, help students appreciate linkages among topics, and can make material more interesting to them. In both cases, this should lead to better retention.

5.5 Concluding Comment

The above represents a set of key guidelines that need to underpin the development of a high-quality software engineering program. These are not necessarily the only concerns. For each institution, there are likely to be local and national needs driven by industry, government, etc. The aspirations of the students themselves also need to be considered. Students must see value in the education, and they must see it meeting their needs; often this is conditioned by their achievements (e.g. what they have been able to build) during their program and by their career aspirations and options. Certainly, they should feel confident about being able to compete internationally, within the global workforce.

Any software engineering curriculum or syllabus needs to integrate all these various considerations into a single, coherent program. Ideally, a uniform and consistent ethos should permeate individual classes and the environment in which the program is delivered. A software engineering program should instill in the student a set of expectations and values associated with engineering high-quality software systems.

Chapter 6: Courses and Course Sequences

In this chapter we present a set of example curricula that can be used to teach the knowledge described in SEEK (Chapter 4) according to the guidelines described in Chapter 5.

This section is organized as follows. Section 7.1 describes how courses are categorized and the coding scheme used. Subsequent sections discuss patterns for introductory courses, intermediate software engineering courses and other courses, respectively. Details of the courses, including mappings to SEEK, are left to Appendix A.

This document is intended as a resource for institutions that are developing or improving programs in software engineering at the undergraduate level, as well as for accreditation agencies that need sample curricula to help them make decisions about various institutions' programs. The patterns and course descriptions describe reasonable approaches to designing and delivering programs and courses, but are not intended to be prescriptive nor exhaustive. It is suggested, however, that institutions strongly consider using this chapter as a basis for curriculum design, since similarity among institutions will benefit at least three groups: 1) students who wish to transfer, 2) employers who wish to understand what students know, and 3) the creators of educational materials such as textbook authors.

Even if an institution decides to base their curriculum on those presented here, it must consider its own local needs, and adapt the curriculum as required. Local issues that will vary from institution to institution include 1) the preparation of the entering students, 2) the availability and expertise of faculty at the institution, 3) the overall culture and goals of the institution, and 4) any additional material that the institution wants its students to learn. Developing a comprehensive set of desired student outcomes for a program (see … - to be added) should be the starting point.

Relationship to CCCS

The CCCS volume contains a set of recommendations for undergraduate programs in Computer Science. While undergraduate degrees in Software Engineering are different from degrees in Computer Science, the two have much in common, particularly at the introductory levels. We will refer to descriptions developed in CCCS when appropriate, and show how some of them can be adopted directly – as will be important for many institutions that offer both computer science and software engineering degrees.

How this section was developed

To develop these curricula, a subcommittee of volunteers created a first draft. Numerous iterations then followed, with changes largely made by steering committee members as a result of input from various workshops. The original committee members started with SEEK, CCCS, and a survey of 32 existing bachelors degree programs from North America, Europe and Australia. A key technique to develop curricula was to determine which SEEK topics can be covered by reusing CCCS courses. A key subsequent step was to work out ways to distribute the remaining SEEK material into cohesive software engineering courses, using the existing programs as a guide. It should be noted that many of the existing bachelors degree programs do not, in fact, cover SEEK entirely, so the proposals do not exactly match any existing program.

6.1 Course Coding Scheme

This document uses the following coding scheme:

XXnnn{-xxxx}

Where:

XX is one of

CS – for courses taken from the CCCS volume
SE – for software engineering courses defined here
NT – for non-technical courses defined here

nnn is an identifying number, where:

- the first digit indicates the earliest year in a four-year program in which the course would typically be taken
- the second digit divides the courses into broad subcategories within SE
  0 means the course is broad, covering many areas of SEEK
  1 means the course has a heavy weight in design and computing fundamentals that are the basis for design
  2 means the course has a heavy weight in process-oriented material
- the third digit distinguishes among courses that would otherwise have the same number

xxxx is an alphabetic mnemonic tag added to most courses codes to help the reader remember the subject matter. It is not an essential part of the numbering scheme since the XXnnn part is the unique identifier.

Except where specified, all courses are '40-hour' standard courses in the North-American model. As discussed earlier, this does not mean that there has to be 40 hours of lecturing, but that the amount of material covered would be equivalent to a traditional course that has 40 hours of lectures, plus approximately double that time composed of self-study, labs, tutorials, exams, etc.

We will also color-code courses according to the following categories.

The first three colors are used to indicate courses that would typically be taught early and represent essential introductory material. Specific courses and sequences of these are discussed in the next section, Section 6.2.

SE+CS introductory courses - first year start

introductory computer science courses from CCCS

Mathematics fundamentals courses

The second group of courses primarily cover core software engineering material from SEEK. These are discussed in Section 6.3.

Software engineering core courses

Capstone project course

The next group of courses cover material that is essential in the curriculum but is neither introductory, nor core software engineering material. Such courses are discussed in Section 6.4

Intermediate fundamental computer science courses

Non-technical compulsory courses

The following pastel colors are used to indicate course categories that will be elective and optional in at least some institutions, while perhaps required in others. These are also discussed in Section 6.4.

Mathematics courses that are not SE core

Technical (SE/CS/IT/CE) courses that are not SE core

Science/engineering courses covering non-SEEK topics

General non-technical courses

Unconstrained

The last category is used when course slots are specified, yet no specific course is specified for the slot.

6.2 Introductory Sequences Covering Software Engineering, Computer Science and Mathematics Material

There are several approaches to introducing software engineering to students in the first year-and-a-half of a bachelors degree program. In this section we briefly describe the sequences and the courses they include. We initially describe sequences that teach introductory computing material, and then we discuss sequences for teaching mathematics.

The distinguishing feature of the two main computing sequences is whether students start with courses that immediately introduce software engineering concepts, or whether they instead start with a pure computer science first year and are only introduced to software engineering in a serious way in second year. There is no clear evidence regarding which of these approaches is best. The CS-first approach is by far the more common, and, for solid pragmatic reasons, seems likely to remain so. However, the SE-first approach is seen by some to better ensure students develop a proper sense of what software engineering is all about. The following are some of the perceived advantages and disadvantages of the two approaches:

Arguments for the SE-first approach:

- Students are taught from the start to think as an engineer, to consider requirements and design before coding, to think about process, and to adopt other software engineering best practices. In other words, they are taught good habits right from the start.
- Computer science courses in many institutions are taught in a way that instills a code-oriented mindset in students, and therefore the bad habit of first thinking in terms of code as opposed to requirements, design, process and the engineering approach. It is felt that this mindset is hard to break later, and leads to students being skeptical of many of the tenets of software engineering. Even though CS first year course designs may list some software engineering concepts to be taught, it is all too easy for instructors not educated as software engineers to downplay these.

Arguments for a CS-first approach

- Programming is a fundamental skill required by all software engineers; it is also a skill that takes much practice to acquire. The more and earlier students practice programming the more competent they are likely to become. (Some would disagree with the importance of programming to a software engineer, but the consensus among those developing this document is that it is an essential skill.)
- Students who know little about computers or programming may not be able to grasp SE concepts in the first year, or would find those concepts have little meaning for them.
- There are many textbooks for standard first-year CS courses, and few that take a truly SE-first approach. Teaching in an SE-first manner might therefore require instructors to produce much of their own material.
- Since many institutions offer both SE and CS degrees, they will want to share courses to reduce resource requirements.
- There is a shortage of SE faculty in many institutions. Those SE faculty available are needed to teach the more advanced courses. Diverting them to teach first year can reduce the quality of later SE courses.
- Most employment open to students after their first year will involve programming. Employers will be reluctant to give students responsibilities for design or requirements until they have matured further. Thus development of programming skills should be emphasized in the first year.

There is clearly some wisdom in both approaches, and little convincing evidence that either is as ‘bad’ or ‘good’ as some people might claim. In order to strike some middle ground, the courses in both sequences do indeed have some material from the ‘other side’. The core CCCS first year courses have a certain amount of SE coverage, while the first-year courses we propose for the SE-first approach do also teach the fundamentals of implementation, although not as deeply as the CS courses.

It is intended that by the time students reach the end of either introductory sequence, they will have covered the same topics.

6.2.1 Introductory Computing Sequence A: Start to software engineering in first year.

In this sequence, a student's first year starts with two courses, SE101 and SE102 (described later) that introduce software engineering in conjunction with some programming and other computer science concepts. These courses differ from traditional introductory computer science courses in two ways: (1) Because of the inclusion of a more in-depth introduction to software engineering, less time is spent on developing programming skills; and (2) The engineering perspective fundamental to software engineering plays a major role in the course. Thus the impact of a few extra hours formally devoted to software engineering is multiplied through an emphasis on using a software engineering approach in all programming assignments.

In the second year, students then take courses CS103 and SE200, which prepare students for the intermediate sequences discussed in Section 6.3. CS103 and SE200 combine to finish the development of basic computing knowledge and programming skills in the students in the program. SE200 contains some of the programming-oriented material normally found in introductory computing courses but not included in SE101 and SE102. CS103 and SE200 can be taken concurrently or either one before the other; for scheduling purposes it will often be best of they are taken at the same time.

SE101 → SE102 → CS103
SE200

The following are brief descriptions for the above courses. Additional details are in Appendix A.

SE101 Introduction to software engineering and computing

A first course in software engineering and computing for the software engineering student who has taken no prior computer science at the university level. Introduces fundamental programming concepts as well as basic concepts of software engineering: requirements, modeling, design, and testing; software engineering as an engineering discipline; problem solving; professional ethics; human factors.

SE102 Software engineering and computing II

A second course in software engineering, delving deeper into software engineering concepts while continuing to introduce computer science fundamentals. Includes coverage of design strategies, verification and validation, software evolution as well as basic principles of programming languages, operating systems and databases, all in the software engineering context. Prerequisite: SE101.

SE200 Software Engineering and computing III

Continues a broad introduction to software engineering and computing concepts, with particular emphasis on modeling and abstraction as used in software architecture, design, and implementation. In depth coverage of UML. Translation of a model into code using a programming language. Introduction to user interface design and project management. Intended for students who will subsequently be taking more advanced SE courses. Prerequisite: SE102

CS103 Data Structures and Algorithms

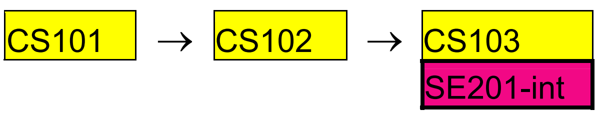
Any variant of CS 103 from the CCCS volume can be used (e.g. those from the imperative-first or objects-first sequences). Normally this course has CS102 as a prerequisite; in this sequence, SE102 is the prerequisite. The description from the CS volume is:

> *Builds on the foundation provided by the CS101$_{I}$-102$_{I}$ sequence to introduce the fundamental concepts of data structures and the algorithms that proceed from them. Topics include recursion, the underlying philosophy of object-oriented programming, fundamental data structures (including stacks, queues, linked lists, hash tables, trees, and graphs), the basics of algorithmic analysis, and an introduction to the principles of language translation.*

See the CS volume for further details. A mapping to SEEK is in Appendix A of this volume.

6.2.2 Introductory Computing Sequence B: Introduction to software engineering in second year

In this sequence, a student starts with one of the initial sequences of computer science courses specified in the CS volume for CS degrees. Specialization in software engineering starts in second year with SE201, which can be taken at the same time as the third CS course.



The CCCS volume offers several variants of the CS introductory courses. Any of these can be used, although the imperative-first (subscript I), and objects-first (subscript O) seem the best as foundations for software engineering. CS103 was described in the last subsection; the imperative-first versions of the first two CS courses, along with SE201-int are briefly described below. Note that CS101 and CS102 cover mostly CMP.cf topics from SEEK, but also cover small amounts of software engineering material from other SEEK knowledge areas. Even with the inclusion of the basics of software engineering, it is not expected that software engineering practices will be strongly emphasized in the programming assignments.

The CCCS volume does allow for a 'compressed' introduction to computer science, in which CS101, CS102 and CS103 are taught instead as a 2-course sequence CS111 and CS112. If such courses are used in software engineering degrees, coverage of SEEK will be insufficient unless either students are admitted with some CS background, or else extra CS coverage is added to other courses.

CS101I Programming Fundamentals

This is a standard introduction to computer science, using an imperative-first approach. The description from the CS volume is:

Introduces the fundamental concepts of procedural programming. Topics include data types, control structures, functions, arrays, files, and the mechanics of running, testing, and debugging. The course also offers an introduction to the historical and social context of computing and an overview of computer science as a discipline.

See the CCCS volume for further details. A mapping to SEEK is in the Appendix A of this volume.

CS102I The Object-Oriented Paradigm

This is the second in a standard sequence of introductory CS courses. The description from the CS volume is:

Introduces the concepts of object-oriented programming to students with a background in the procedural paradigm. The course begins with a review of control structures and data types with emphasis on structured data types and array processing. It then moves on to introduce the object-oriented programming paradigm, focusing on the definition and use of classes along with the fundamentals of object-oriented design. Other topics include an overview of programming language principles, simple analysis of algorithms, basic searching and sorting techniques, and an introduction to software engineering issues.

See the CCCS volume for further details, and for the object-first variants. A mapping to SEEK is in Appendix A of this volume.

SE201-int Introduction to Software Engineering for Software Engineers

Presents the basic principles and concepts of software engineering. This course gives broad coverage of the most important terminology and concepts in software engineering. It is designed for students who will be subsequently taking more advanced software engineering courses. Upon completing this course, students will be able to do basic modeling and design, particularly using UML. They will also have a basic understanding of requirements, software architecture, and testing. Prerequisite CS102

6.2.3 Introductory Mathematics Sequences

Discrete mathematics is the mathematics underlying all computing, including software engineering. It has the importance to software engineering that calculus has in other branches of engineering. Statistics and empirical methods are also of key importance to software engineering.

The mathematics fundamentals courses cover SEEK's FND.mf topic and some of FND.ef – i.e. discrete mathematics plus probability, statistics and empirical methods. We have reused CCCS courses CS105 and CS106. Since the CCCS volume lacks an appropriate course for empirical and statistical material, MA271-sta was created to cover statistics and empirical methods.

It is highly recommended that the discrete mathematics courses be taught starting in first year in lieu of any other mathematic course requirements since it is more important that a strong discrete mathematic foundation is made than, for example, calculus. It is not strictly necessary, however, since this material is needed for most, but not all, of the intermediate software engineering courses discussed in the next section.

CS105-ds1 → CS106-ds2 → MA271-sta

CS105 Discrete Structures I

Standard first course in discrete mathematics. Taught in a way that shows how the material can be applied to software and hardware design. The description from the CS volume is as follows:

> *Introduces the foundations of discrete mathematics as they apply to computer science, focusing on providing a solid theoretical foundation for further work. Topics include functions, relations, sets, simple proof techniques, Boolean algebra, propositional logic, digital logic, elementary number theory, and the fundamentals of counting.*

See the CCCS volume for more details.

CS106 Discrete Structures II

Standard second course in discrete mathematics. The description from the CS volume is as follows:

> *Continues the discussion of discrete mathematics introduced in CS105. Topics in the second course include predicate logic, recurrence relations, graphs, trees, matrices, computational complexity, elementary computability, and discrete probability.*

See the CCCS volume for more details.

MA271-sta Statistics and Empirical Methods

Applied probability and statistics in the context of computing. Experiment design and the analysis of results. The course is taught using examples from software engineering and other computing disciplines. Prerequisite or co-requisite: CS 106.

6.3 Core Software Engineering Sequences

In this section, we present two sequences, each containing six intermediate software engineering courses. We also present the capstone course. None of the courses in these sequences is fully specified (i.e. none has all the 40 hours allocated to topics). This allows institutions and instructors to be flexible as they adapt the courses to their needs.

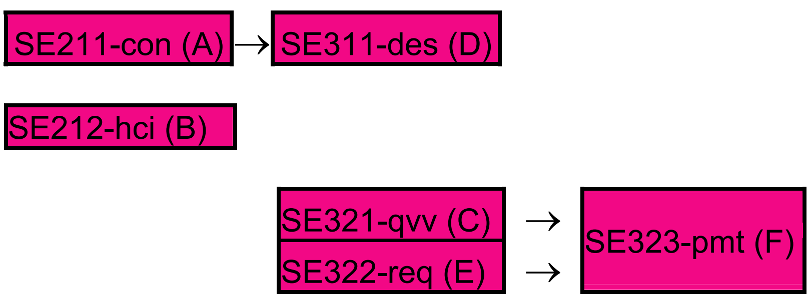
Both 6-course sequences follow either SE201-int or SE 200, and would normally be started in the second year. The sequences cover much of the core SE material in SEEK. Both group the material in a slightly different way, but ultimately result in the same knowledge being taught.

In both sequences, the courses are labelled (A), (B) … (F). These letters are used in the course patterns discussed in section 6.5; they indicate the slots into which the courses can be placed.

Indentation from the left margin means that a course should not be taken too early in the curriculum since it requires maturity, but that there is no explicit prerequisite

Note that SE212-hci is found in both packages.

6.3.1 Core Software Engineering Package 1



The following are descriptions of the courses in this package. Additional details, including a mapping to SEEK, can be found in the appendix.

SE211-con Software Construction

Basics of software construction, including underlying formal approaches and the mathematics relating to those approaches. State-based construction techniques, run-time configuration, grammar-based input processing, basics of concurrency and distributed software; use of middleware. Prerequisites: SE201-int or SE 200, CS103, CS105-ds1.

SE212-hci Software Engineering Approach to Human Computer Interaction

A comprehensive introduction to the principles and techniques of human-computer interaction and user interface design, with a focus on highly usable software. User and task modeling, user centered design; evaluation of user interfaces; detailed discussion of many UI design issues such as use of coding techniques (color, icons, sound, etc.), screen and web page design, feedback and error messages, internationalization of user interfaces, response time, accessibility to the disabled; user interfaces for different types of devices; voice user interfaces, etc. This course will require students to implement user interfaces, but the focus must not be on UI tools and technologies themselves. Prerequisites: CS103; Pre- or Co-Requisites: SE201-int or SE200.

SE311-des Software Design and Evolution

Advanced software design, particularly aspects relating to distributed systems and software architecture. Evaluation and evolution of designs. Prerequisite: SE211-con.

SE321-qvv Quality, verification and validation

Quality: how to assure it and verify it.. Avoidance of errors and other quality problems. Reviews, testing. Quality process standards. Product and process assurance. Prerequisites: SE201-int or SE200, , plus at least one additional software engineering course at the 2 level or higher.

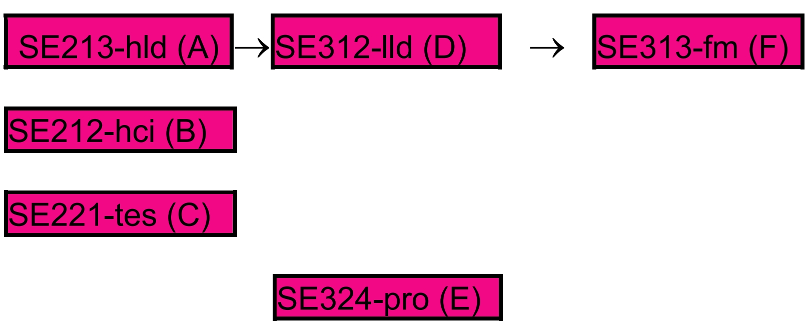
SE322-req Requirements

In-depth course about software requirements: Types of models, eliciting requirements, specification and documentation, requirements validation, requirements management. Prerequisites: SE201-int or SE200, , plus at least one additional software engineering course at the 2 level or higher

SE323-pmt Project Management

In-depth course about project management. It is assumed that by the time students take this course they will have a broad and deep understanding of other aspects of software engineering. Process concepts and implementation; management concepts; project planning and control; software personnel management; configuration management. Prerequisites: SE321-qvv, SE322-req.

6.3.2 Core Software Engineering Package 2



Note that SE212-hci has already been discussed in the context of Package 1.

SE213-hld Design and Architecture of Large Software Systems

Modeling and design of large-scale, evolvable systems; managing and planning the development of such systems – including the discussion of configuration management; software architecture. Prerequisites: SE200 or SE201, CS103

SE221-tes Testing

In-depth course on all aspects of testing, as well as other aspects of verification and validation, including specifying testable requirements, reviews and product assurance. Prerequisites: SE201-int or SE200

SE312-lld Low-Level Design

Techniques for low-level design and construction, including formal approaches. Detailed design for evolvability. Prerequisite: SE212-hld

SE324-pro Process and Management

Software processes in general; requirements processes and management; evolution processes; quality processes; project personnel management; project planning. Prerequisites: SE201-int or SE 200, plus at least two additional software engineering courses at the 2 level or higher.

SE313-fm Formal Methods in Software Engineering

Approaches to software design and construction that employ mathematics to achieve higher levels of quality. Mathematical foundations of formal methods; formal modeling; validation of formal models; formal design analysis; program transformations. Prerequisites: SE2315-des2, SE325-pro2, CS106-ds2.

6.3.3 Software Engineering Capstone Project

As has been discussed in the guidelines presented in the last chapter, a capstone project course is essential in a software engineering degree program. We highly recommend that it be a full-year course (80 lecture-equivalent hours).

The capstone course provides students with the opportunity to undertake a significant software engineering project, in which they will deepen their knowledge of many SEEK areas. It should cover a full-year (i.e. 80 lecture-equivalent-hours). It covers a few hours of a variety of SEEK topics since it is expected that students will learn some material on their own during this course, and will deepen their knowledge in several areas to the ‘a’ level of Bloom’s taxonomy.

SE400-cap

SE400-cap Software Engineering Capstone Project

Provides students, working in groups, with a significant design experience in which they can integrate much of the material they have learned in their program, including matters relating to professionalism and project management. The project will ideally involve a real-world customer, but will be supervised by a faculty member. This course would normally not involve any formal lectures, except for co-ordination purposes. Students would be expected to present their work regularly to other students. Prerequisites: At least 5 software engineering courses at the 2 level or above. Pre or Co-Requisites: SE323-pmt or SE324-pro

6.4 Completing the Curriculum: Additional Courses

The introductory and core SE courses discussed in the last two sections cover much of the required material, but there are still several categories of courses remaining to discuss:

6.4.1 Courses covering the remaining compulsory material

Intermediate fundamental computer science courses

The intermediate fundamental computer science courses are CCCS courses in the 200 series, and cover much of the remaining CMP.cf topics. Any curriculum covering SEEK will need at least two of these; the patterns in the next section all have three selected courses, but that illustrates only one possible approach. Some curricula, not shown here, may want to spread the intermediate SEEK CMP.cf material out over more than three courses.

See the computer science volume for sample courses in this category. Mappings of some courses to SEEK can be found in Appendix A to this document.

Non-technical compulsory courses

The non-technical compulsory courses primarily cover the FND.ec topic and the PRF area of SEEK – i.e. engineering economics, communication and professionalism. Although it would be possible to compress the necessary SEEK material into a single course, we have shown the material spread three courses so it can be covered in more depth.

NT271-eco Engineering Economics

This is a standard engineering economics course as taught in many universities. A relatively small fraction of this course is actually required by SEEK, but it would be desirable for software engineering students to learn more than that minimum.

NT181-com Group Dynamics and Communication

Communication skills are highly regarded in the software industry but they are also fundamental to success in collegiate careers. This course should provide the necessary basis and the practice to make the students comfortable in the area.

NT291-eth Professional Software Engineering Practice

Professional Practice is concerned with the knowledge, skills, and attitudes that software engineers must possess to practice software engineering in a professional, responsible, and ethical manner. It is anticipated that a wide variety of additional material may be taught in this course. A technique that has worked well is to employ guest speakers from professional societies. See also CCCS CS280.

Introductory Computing Sequence

This is a reference to either the A (SE101, SE102, CS103, and SE200) or the B (CS101, CS102, CS103, and SE201-int) sequence as defined in section 6.2.

6.4.2 Non-SEEK courses

Curriculum slots designated non-SEEK cover material outside the scope of SEEK. We have included several of them in example curricula to assist curriculum designers develop programs that cover more than just SEEK. A certain number of such courses are essential for any interesting and well-rounded SE program. Curriculum designers and/or students have the flexibility to make their own choices based on their institutional or personal needs, or based on the needs of accreditation agencies that look for a broader engineering, science or humanities background.

All courses in these categories are shown in italics with light background colors.

Mathematics courses that are not SE core

These cover two types of mathematics courses: a) material such as calculus that is not required for a software engineering program according to SEEK, but is nonetheless required in many curricula for various reasons; b) elective mathematics courses. We show sample course sequences containing such courses.

Most universities, especially in North America, will teach calculus, often in first year. SEEK does not contain calculus, because it is not used by software engineers except when doing domain-specific work (e.g. for other engineers or for scientists) and hence is not essential for *all* software engineering programs. However, there are a number of reasons why most programs will include calculus: 1) It is believed to help encourage abstract thinking and mathematical thinking in general; 2) Although needed in the workplace by only a small percentage of software engineers, it is just not readily learned in the workplace.

Other mathematics commonly found in SE curricula are linear algebra and differential equations.

Technical (SE/CS/IT/CE) courses that are not SE core

These courses, cover technical material beyond the scope of the essential SEEK topics. Such courses could be compulsory in a particular program or electives chosen by students. They might cover topics in SEEK in greater depth than SEEK specifies, or else might cover material not listed in SEEK at all. This chapter does not give detailed specifications of such courses, but slots are shown in the course patterns. The reader can consult the Computer Science, Information Systems or Computer Engineering volumes for examples.

Science/engineering courses covering non-SEEK topics

These cover material such as physics, chemistry, electrical engineering, etc. Most software engineering programs, especially in North America, will include some such courses, particularly physics courses.

The rationale for including science courses is that they give students experience with the scientific method and experimentation. Similarly, taking other engineering courses expands students' appreciation for engineering in general. Taking some science and engineering courses will also help students who later on want to develop software in those domains.

Courses in this category are not specified in further detail in this document.

General non-technical courses

These slots are for courses in business, social sciences, humanities, arts etc. Most programs will make some such courses compulsory, particularly in the US, where there is a tradition of requiring some 'liberal arts'. Some universities will want to incorporate specific streams of non-technical courses, e.g. a stream of business courses.

6.5 Curriculum Patterns

In this section we present some example patterns showing how the courses described in the last three sections can be arranged in a degree program along with additional non-core courses. One general pattern is presented as the recommended structure of a software engineering program.

All of the example patterns should be seen as examples; they are not intended to be prescriptive (unlike SEEK). They illustrate approaches to packaging SEEK topics in various contexts.

The main features that differentiate the example patterns are:

- The international context
- The computer science or engineering school context
- Whether software engineering is to be taught starting in first year or second
- Whether there are two semesters per academic year, or three quarters.

There is considerable flexibility in the intermediate fundamental CS courses; a set of CCCS courses that cover appropriate areas of SEEK is suggested.

We have included three non-technical courses to cover relevant areas of SEEK. We suggest starting with a communications course (e.g NT181-com) very early, and deferring the ethics cse (e.g. NT291-eth) as shown until students gain more maturity. Many variations are, however, possible, including rolling the SEEK material in these courses into one or two courses instead of three.

The discrete math courses are taught in the first year, with Calculus I and II shown as taught in the second year. The main argument in favor of this arrangement is that the discrete math courses are to software engineering what calculus is to the rest of engineering, and therefore should be taught early to form a foundation. However, some institutions may wish to start with calculus and either teach discrete math concurrent with or consecutive to it. It is recognized that teaching calculus first allows SE programs to mesh with existing CS programs; it also ensures that SE students take calculus in classes with other students of the same age group.

Pattern SE - Recommended General Structure

Year1		Year 2		Year 3		Year 4	
Sem 1A	Sem 1B	Sem 2A	Sem 2B	Sem 3A	Sem 3B	Sem 4A	Sem 4B
Intro	*computing*	*sequence*	CS	CS	CS	SE400-cap	SE400-cap
CS105-ds1	CS106-ds2	*Calc 1*	*Calc 2*	MA271-sta		SE	*Tech elective*
NT		SE200/201	SE	SE	SE	*Tech elective*	*Tech elective*
		NT	SE	NT	*Tech elective*		

The remaining chapter is devoted to illustrating specific instances of applying Pattern SE in varying contexts.

Pattern N2S-1 - North American Year-2-Start with Semesters

This pattern illustrates one way that courses can be arranged that should be widely adaptable to the needs of many North American universities operating on a semester system. Many course slots are left undefined to allow for adaptation. Two example adaptations are shown later.

The pattern starts its technical content with CS101, CS102 and CS103   The pattern also has SE201-int taken in parallel with CS103 (see above for discussion of this sequence). The SE101, SE102, CS103, SE200 sequence could be substituted.

Following the introductory course SE201-int (or SE200), students would take one of the packages of six SE courses described above that cover specific areas in depth.

Pattern N1S - US model using introductory computing sequence A (starting SE early)

This model shows the use of the first-year-start sequence: SE101, SE102 and SE200. It represents how an institution might build a typical software engineering program in a software engineering context.

Year1		Year 2		Year 3		Year 4	
Sem 1A	Sem 1B	Sem 2A	Sem 2B	Sem 3A	Sem 3B	Sem 4A	Sem 4B
SE101	SE102	CS103	CS270T-db	CS220-arc	SE D	CS226-os-nt	SE400-cap
Calc 1	*Calc 2*	SE200	SE212-hci	SE A	SE E	SE400-cap	*Tech elect.*
CS105-ds1	*CS106-ds2*	*Physics 1*	MA271-sta	SE C	*Tech elect.*	SE F	*Tech elect.*
Gen ed	*Gen ed*	NT181-com	*Physics 2*	*Sci Elective*	NT291-eth	*Gen ed*	*Gen ed*
Gen ed	*Gen ed*	*Psychology*	*Sci Elective*	*Sci Elective*	*Gen ed*		

Pattern N2S-1c - in a computer-science department

The pattern shown below is typical of a software engineering program that might be built in a computer science context. Such programs may have evolved from computer science programs or may require co-existence with a computer science program.

Year1		Year 2		Year 3		Year 4	
Sem 1A	Sem 1B	Sem 2A	Sem 2B	Sem 3A	Sem 3B	Sem 4A	Sem 4B
CS101	CS102	CS103	CS220-arc	CS226-os-nt	CS270T-db	SE400-cap	SE400-cap
CS105-ds1	CS106-ds2	*Calc 1*	*Calc 2*	MA271-sta	SE D	SE F	*Tech elective*
NT181-com	*Linear Alg*	SE201-int	SE A	SE C	SE E	*Tech elective*	*Tech elective*
Physics	*Any science*	NT271-eco	SE212-hci	NT291-eth	*Tech elective*	*Tech elective*	*Tech elective*
Gen ed	*Gen ed*		*Gen ed*	*Gen ed*	*Gen ed*	*Gen ed*	*Gen ed*

Pattern N2S-1e - in an engineering department

Programs in a North American engineering department typically begin with a rigorous calculus sequence (three semesters), linear algebra, probability and statistics, physics and chemistry. Introductory courses in other areas of engineering are given during the first year. For SE programs in EE or CE departments, circuits and electricity are common. Programming for engineers is usually required in the first year. The introductory computer science sequence is often the compressed CS111, CS112 (CCCS) sequence, although we have maintained the 3-course sequence below since we believe this is much better for software engineers.

Year1		Year 2		Year 3		Year 4	
Sem 1A	Sem 1B	Sem 2A	Sem 2B	Sem 3A	Sem 3B	Sem 4A	Sem 4B
CS101	CS102	CS103	CS220-arc	CS226-os-nt	CS270T-db	SE400-cap	SE400-cap
Calc 1	*Calc 2*	CS106-ds2	*Linear Alg*	MA271-sta	SE D	SE F	*Tech elective*
NT181-com	CS105-ds1	SE201-int	SE A	SE C	SE E	*Tech elective*	*Tech elective*
Physics 1	*Physics 2*	NT271-eco	SE212-hci	NT291-eth	*Tech elective*	*Tech elective*	*Tech elective*
Chemistry	*Engineering*	*Calc 3*	*Gen ed*	*Gen ed*	*Gen ed*	*Gen ed*	*Gen ed*

Pattern E-1 - Compressed model for a country in which it is assumed calculus and science is not needed or is taught in high school, and less general education is needed

Some countries, including most of the UK, have secondary school systems that bring students to a higher level of science and mathematics. Such systems also tend to have very focused post-secondary education, requiring much less in the way of general education (humanities etc.). The following pattern shows one way of teaching SE in those environments.

Year1		Year 2		Year 3	
Term 1A	Term 1B	Term 2A	Term 2B	Term 3A	Term 3B
CS101	CS102	CS103	CS merged	SE400-cap	SE400-cap
CS105-ds1	CS106-ds2	MA271-sta	SE D	SE F	*Tech elective*
NT181-com	SE201-int	SE A	SE E	*Tech elective*	*Tech elective*
NT271-eco	NT291-eth	SE C	SE212-hci	*Tech elective*	*Tech elective*

Pattern E-2 – Another model for a country where calculus and science is not needed.

This pattern also illustrates the use of SE101 and SE102, as well as the delay of some of the core SE courses until students have gained maturity.

Year1		Year 2		Year 3		Year 4	
Sem 1A	Sem 1B	Sem 2A	Sem 2B	Sem 3A	Sem 3B	Sem 4A	Sem 4B
SE101	SE102	CS103	SE200	SE A	SE212-hci	SE D	SE F
CS overview	CS106-ds2	CS220-arc	CS226-os-nt	*Tech elect.*	SE C	SE E	SE400-cap
CS105-ds1	MA271-sta	NT291-eth	CS270T-db	*Tech elect.*	*Tech elect.*	SE400-cap	*Tech elect.*
NT181-com							

Pattern N3Q-1 - North American year 3 start with quarters

Some North American universities operate on a quartered system, with three quarters instead of two semesters. The following pattern accommodates this, assuming that four courses are taught each quarter. This pattern also illustrates one way of delaying the SE core courses until third year.

Year 1			Year 2		
Quarter 1A	Quarter 1B	Quarter 1C	Quarter 2A	Quarter 2B	Quarter 2C
CS101	CS106-ds2	CS 102	CS 103	CS270T-db	CS226-os-nt
CS105-ds1	*Chemistry*	*Math*	CS220-arc	*Calc 2*	*Calc 3*
Physics 1	*Physics 2*	*Engineering*	*Calc 1*	NT291-eth	*Gen ed*
Gen ed	NT181-com	*Gen ed*	*Math*		

Year 3			Year 4		
Quarter 3A	Quarter 3B	Quarter 3C	Quarter 4A	Quarter 4B	Quarter 4C
SE201-int	SE A	SE D	cap1	cap2	cap3
SE212-hci	SE C	SE E	SE F	*Tech elect.*	*Tech elect.*
MA271-sta	*Tech elect.*	*Gen ed*	*Tech elect.*	*Gen ed*	*Gen ed*
NT181-com			*Gen ed*		

Chapter 7: Adaptation to alternative environments

Software engineering curricula do not exist in isolation. They are found in institutions and these institutions have differing environments, goals, and practices. International issues are not the only problem curriculum implementers will experience. Software engineering curricula must be able to be delivered in a variety of fashions and to be part of many different types of institutions.

There are two main categories of "alternative" environments that will be discussed in this section. The first is the alternative *teaching* environment. These environments use non-standard delivery methods. The second is the alternative *institutional* environment. These institutions differ in some significant fashion from the usual university.

7.1 Alternative teaching environments

As higher education has become more universal, the standard teaching environment has tended toward an instructor in the front of a classroom. Although some institutions still retain limited aspects of a tutor-student relationship, the dominant delivery method in most higher education today is classroom type instruction. The instructor presents material to a class using lecture or lecture/discussion presentation techniques. The lectures may be augmented by appropriate laboratory work. Class sizes range from fewer than 10 to more than 500.

Instruction in the computing disciplines has been notable because of the large amount of experimentation with delivery methods**.** This may be the result of the instructors' familiarity with the capabilities of technology**.** It may also be the result of the youthfulness of the computing disciplines**.** Regardless of the cause, there are numerous papers in the SIGCSE Bulletin, the Proceedings of the SIGCSE (Special Interest Group in Computer Science Education) annual symposia, the proceedings of the CSEE&T (Conference on Software Engineering Education and Training) conferences, and similar forums, that recount significant modifications to the conventional lecture and lecture/discussion based classrooms. Examples include all laboratory instruction, use of electronic whiteboards and tablet computers, problem based learning, role-playing, activity based learning, and various studio approaches that integrate laboratory, lecture and discussion. As has been mentioned elsewhere in this report, it is imperative that experimentation and exploration be a part of any software engineering curriculum. Necessary curriculum changes are difficult to implement in an environment that does not support experimentation and exploration. A software engineering curriculum will rapidly become out of date unless there is a conscious effort to implement regular change.

Much recent curricular experimentation has focused on "distance" learning. The term is not well defined. It applies to situations where students are in different physical locations during a scheduled class. It also applies to situations where students are in different physical locations and there is no scheduled class time. It is important to distinguish these two cases. It is also important to recognize other cases as well, for example the situation where students cannot attend regularly scheduled classes.

7.1.1 Students at different physical locations

Instructing students at different physical locations is a problem that has several solutions. Audio and video links have been used for many years and broadband Internet connections are less costly and more accessible. Instructor-student interaction is possible after all involved have learned how to manage it without confusion. Two-way video makes such interaction almost as natural as the interaction in a self-contained classroom. On-line databases of problems and examples can be used to further support this type of instruction. Web resources, email, and Internet chat can provide a reasonable instructor "office hour" experience. Assignments can be submitted by email or by using a direct Internet connection. The current computing literature and departmental Web sites contain numerous descriptions of "distance learning" techniques.

It should be noted that a complete solution to the problem of delivering courses to students in different locations is not a trivial matter and any solution that is designed will require significant planning and appropriate additional support. Some may argue that there is no need to make special provision for added time and support costs when one merely increases the size of an existing class by adding some "distance" students. Experience indicates that this is always a very poor idea.

Students in software engineering programs need to have experience working in teams. Students who are geographically isolated need to be accommodated in some fashion. It is unreasonable to expect that a geographically separated team will be able to do all of its work using email, chat, blogs and newsgroups. Geographically separated teams need additional monitoring and support. Videoconferencing and teleconferencing should be considered. Instructors may also want to schedule some meetings with the teams, if distances make this feasible. Beginning students require significantly more monitoring than advanced students because of their lack of experience with geographically separated teams.

One other problem with geographically diverse students is the evaluation of student performance. Appropriate responsible parties will need to be found to proctor examinations and check identities of examinees. Care should be taken to insure that evaluation of student performance is done in a variety of ways. Placing too much reliance on one method (e.g., written examinations) may make the evaluations unreliable.

7.1.2 Students in class at different times

Some institutions have a history of providing instruction to "mature" students who are employed in a full-time job. Because of their work obligations, employed students are often unable to attend regular class meetings. Videotaped lectures, copies of class notes, and electronic copies of class presentations are all useful tools in these situations. A course Web site, a class newsgroup, and a class distribution list can provide further support.

There is also instruction that does not have any scheduled class meetings. Self-scheduled and self-paced classes have been used at many institutions. Classes have also been designed to be completely "Web-based." Commercial and open-source software has been developed to support many aspects of self-paced and Web-based courses. Experience shows that the development of self-paced and Web-based instructional materials is very expensive and very time consuming.

Students who do not have scheduled classroom instruction will still need team activities and experiences. Many of the comments made above about geographically diverse teams will also apply to them. An additional problem is created when students are learning at wildly different rates. Because content will be covered at different times by different students, it is not feasible to have content instruction and projects integrated in the same unit. Self-paced project courses are another serious problem. It will be difficult to coordinate team activities when different team members are working at different paces.

7.2 Curricula for Alternative Institutional Environments

7.2.1 Articulation problems

Articulation problems arise when students have taken one set of courses at one institution or in one program and need to apply these to meet the requirements of a different institution and/or program.

If software engineering curricula existed all alone there would be no articulation problems. But this is rarely the case. Software engineering programs exist in universities with multiple colleges, schools, divisions, departments and programs. Software engineering programs exist in universities that cooperate and compete with other universities and institutions. Some secondary schools offer university level instruction and students expect to receive appropriate credit and placement. Satisfactory completion of a curriculum must be certified when the student has taken classes in different areas of the university as well as at other institutions. Software engineering programs must be designed and managed so that articulation problems are minimized. This means that the internal and external environment at the institution must be considered when designing a curriculum.

7.2.2 Coordination with other university curricula

Many of the core classes in a software engineering curriculum could also be core classes in another curriculum. An introductory computer science course could be required for the curricula in computer science, computer engineering, and software engineering. Certain architecture courses might be part of curricula in computer science, computer engineering, software engineering, and electrical engineering. Mathematics courses could be required for curricula in mathematics, computer science, software engineering, and computer engineering. A project management course may be required by software engineering and management information systems. Upper level software engineering courses could be taken as part of computer science or computer engineering programs. In most universities there will be pressure to have courses do “double duty” whenever possible.

Courses that are a part of more than one curriculum must be carefully designed. There is great pressure to include everything of significance to all of the relevant disciplines. This pressure must be resisted. It is impossible to satisfy everyone’s desires. Courses that serve two masters will inevitably have to omit topics that would be present were it not for the other master. Curriculum implementers must recognize that perfection is impossible and impractical. The minor content loss when courses are designed to be part of several curricula is more that compensated for by the experience of interacting with students with other ideas and background. Indeed, a case can be made that such experiences are so important in a software engineering curriculum that special efforts should be made to create courses common to several curricula.

7.2.3 Cooperation with other institutions

In today’s world, students complete their university education via a variety of pathways. While many students attend just one institution, there are substantial numbers who attend more than one. For a wide variety of reasons, many students begin their baccalaureate degree program at one institution and complete it at another. In so doing, students may change their career goals or declared majors, may move from a liberal arts program to an engineering or scientific program, may satisfy interim program requirements at one institution, may engage in work-related experiences, or may be coping with financial, geographic or personal constraints.

Software engineering curricula must be designed so that these students are able to complete the program without undue delay and repetition through recognition of comparable coursework and aligned programs. It is straightforward to grant credit for previous work (whether in another department, school, college or university) when the content of the courses being compared is substantially identical. There are problems, however, when the content is not substantially similar. While no one wants a student to receive double credit for learning the same thing twice, by the same token no one wants a student to repeat a whole course merely because a limited amount of content topic was not covered in the other course. Faculty do not want to see a student's progress unduly delayed because of articulation issues; therefore, the wisest criteria to use when determining transfer and placement credit are whether the student can reasonably be expected to 1) address any content deficiencies in a timely fashion and 2) succeed in subsequent courses.

To the extent that course equivalencies can be identified and addressed in advance via an articulation agreement, student interests will best be served. Many institutions have formal articulation agreements with those institutions from which they routinely receive transfer students. For example, such agreements are frequently found in the United States between baccalaureate-degree granting institutions and the associate-degree granting institutions that send them transfer students. Other examples can be seen in the 3-2 agreements in the United States between liberal arts and engineering institutions; these agreements allow a student to take three years at a liberal arts institution and two years at an engineering institution, receiving a Bachelor of Arts degree and a Bachelor of Science degree.

The European Credit Transfer System is another attempt to reduce articulation problems in that continent.

7.2.4 Programs for Associate-Degree Granting Institutions in the United States and Community Colleges in Canada

In the United States, as many as one-half of the baccalaureate graduates will have initiated their studies in associate-degree granting institutions. For this reason, it is important to outline a software engineering program of study that can be initiated in the two-year college setting specifically designed for seamless transfer into an upper division (years 3 and 4) program. Regardless of their skills upon entry into the two-year college, students must complete the coursework in its entirety to well-defined competency points to ensure success in the subsequent software engineering coursework at the baccalaureate level. For some students this may require more than two years of study at the associate level. But regardless of this, the goal is the same: to provide a program of study that prepares the student for the upper level institution.

The following is a recommended software engineering program of study for implementation by associate-degree granting institutions. Students who complete this program could reasonably expect to transfer into the upper division program at the baccalaureate institution. Although designed with the United States in mind, certain colleges in Canada and other countries may very well be able to adopt a similar approach.

Proposed Software Engineering Technical Core for North American Community Colleges

For the CS courses listed below, see the Two-Year College Computer Science 2002 report

Computing courses

The three-course sequence

CS101I – Programming Fundamentals
CS102I – The Object-Oriented Paradigm
CS103I – Data Structures and Algorithms

Or the three-course sequence

CS101O – Introduction to Object-Oriented Programming
CS102O – Objects and Data Abstraction
CS103O – Algorithms and Data Structures

SE201-int – Introduction to Software Engineering for Software Engineers

Institutions may also elect to create a software engineering curriculum based on the SE-specific courses (SE101, SE102, CS103, SE200) outlined elsewhere in this report

Mathematics courses

CS105 – Discrete Structures I
CS106 – Discrete Structures II

The following are to articulate with typical university requirements, and do not cover core SEEK material

Calculus I
Calculus II

See also the baccalaureate institution for requirements; some institutions may require linear algebra and/or differential equations

Laboratory Science courses

Two courses in lab science for articulation with most baccalaureate programs.
Recommended: Two physics courses, or one physics plus one chemistry course.

General Education

Students also complete first-year and second-year General Education requirements along with software engineering technical core.

7.2.5 Special programs

Because software engineering is such a new discipline there is a significant demand for certain types of special programs. Some people want to "retrain" in a new field. Others already have a degree in a related field and want a "post-graduate diploma" in software engineering. The curricula for such programs must take into account the previous education of the students as well as their career goals.

It would be foolish to attempt to cram a whole undergraduate curriculum in software engineering into a short retraining program or a one-year post-graduate program. Such an effort does not serve the needs of these students. These programs are best when they have appropriate entrance standards that require at least some practical experience. When this is the case, the students are usually highly motivated. Such students are able to have their experience serve as a reasonable substitute for some of the content that would normally be a part of an undergraduate curriculum.

Chapter 8: Program Implementation and Assessment

Material for this chapter is still under development.

Bibliography for Software Engineering Education

[Abelson 1985] Harold Abelson and Gerald Jay Sussman with Julie Sussman. *Structure and Interpretation of Computer Programs*. Cambridge, MA: MIT Press, 1985.

[ABET 2000] Accreditation Board for Engineering and Technology. *Accreditation policy and procedure manual.* Baltimore, MD: ABET, Inc., November 2000. (http://www.abet.org/images/policies.pdf)

[ACM 1965] *ACM Curriculum Committee on Computer Science. An undergraduate program in computer science—preliminary recommendations.* Communications of the ACM, 8(9):543-552, September 1965.

[ACM 1968] *ACM Curriculum Committee on Computer Science. Curriculum '68: Recommendations for the undergraduate program in computer science.* Communications of the ACM, 11(3):151-197, March 1968.

[ACM 1978] *ACM Curriculum Committee on Computer Science. Curriculum '78: Recommendations for the undergraduate program in computer science*. Communications of the ACM, 22(3):147-166, March 1979.

[ACM 19989] ACM Task Force on the Core of Computer Science, "Computing as a Discipline", *Communications of the ACM*, Vol 32, No 1, January 1989, pp. 1-5.

[ACM 1998] ACM/IEEE-CS Joint Task Force on Software Engineering Ethics and Professional Practices, *Software Engineering Code of Ethics and Professional Practice*, Version 5.2, http://www.acm.org/serving/se/code.htm,September 1998.

[ACM 1999] *ACM Two-Year College Education Committee. Guidelines for associate-degree and certificate programs to support computing in a networked environment.* New York: The Association for Computing Machinery, September 1999.

[ACM 2001] ACM/IEEE-Curriculum 2001 Task Force, *Computing Curricula 2001, Computer Science* , December 2001. (http://www.computer.org/education/cc2001/final/index.htm)

[Andrews and Lutfiyya, 2000] J.H. Andrews and H.L. Lutfiyya *Experiences with a Software Maintenance Project Course,* IEEE Transactions on Education, vol 43, no 4, 383 - 388, November 2000.

[APP 2000] Advanced Placement Program. Introduction of Java in 2003-2004. The College Board, December 20, 2000. (http://www.collegeboard.org/ap/computer-science)

[Bagert 1999] Donald Bagert, Thomas B. Hilburn, Gregory Hislop, Michael Lutz, Michael McCracken, *Guidelines for Software Engineering Education*, Version 1.0, CMU/SEI-99-TR-032, Software Engineering Institute, Carnegie Mellon University, 1999.

[Barnes 1998] B. Barnes, G. Engel, M. Griss, R. LeBlanc, T. Wasserman, L. Werth, "Draft Software Engineering Accreditation Criteria", *Computer*, 31, 4, 73-75, April 1998.

[Bauer 1972] F. L. Bauer, "Software Engineering", *Information Processing*, 71, 1972

[BCS 1989a] British Computer Society and The Institution of Electrical Engineers. *Undergraduate curricula for software engineers*. London, June 1989.

[BCS 1989b] British Computer Society and The Institution of Electrical Engineers. Software in safety-related systems. London, October 1989.

[Beidler et al, 1985] John Beidler, Richard Austing, and Lillian Cassel. Computing programs in small colleges. Communications of the ACM, 28(6):605-611, June 1985.

[Bennett 1986] W. Bennett. A position paper on guidelines for electrical and computer engineering education. IEEE Transactions in Education, E-29(3):175-177, August 1986.

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

[Bourque 2001] P. Bourque and R. Dupuis, eds. *Guide to the Software Engineering Body of Knowledge*, IEEE CS Press, Los Alamitos, CA., 2001.

[Borstler 2002] Jurgen Borstler, David Carrington, Gregory W. Hislop, Susan Lisack, Keith Olson, and Laurie Williams *Teaching PSP: Challenges and Lessons Learned* , IEEE Software, vol 19 , no. 5, September / October, 42 - 48, 2002.

[Bott 1991] Frank Bott, Allison Coleman, Jack Eaton, and Diane Rowland. Professional issues in software engineering. London: Pitman, 1991.

[Brooks 95] Fred P. Brooks, *The Mythical Man-Month, Essays on Software Engineering, Anniversary Edition*. Reading, MA: Addison-Wesley, 1995.

[Burnell 2002] Lisa J. Burnell, John W. Priest, and John R. Durrett, *Teaching Distributed Multidisciplinary Software Development*, IEEE Software, vol 19 , no. 5, September / October, 86 – 93, 2002.

[Buxton 1970] J. N. Buxton and B. Randell (editors) *Software Engineering Techniques*, report of a conference sponsored by NATO Science Committee (Rome, 27 31 October, 1969), 1970.

[Carnegie, 1992] Carnegie Commission on Science, Technology, and Government. Enabling the future: Linking science and technology to societal goals. New York: Carnegie Commission, September 1992.

[Cheston 2002] Grant A. Cheston and Jean-Paul Tremblay *Integrating Software Engineering in Introductory Computing Courses*, IEEE Software, vol 19 , no. 5, September / October, 64 – 71, 2002.

[COSINE, 1967] *COSINE Committee. Computer science in electrical engineering.* Washington, DC: Commission on Engineering Education, September 1967.

[CSAB 1986] *Computing Sciences Accreditation Board. Defining the computing sciences professions.* October 1986. (http://www.csab.org/comp_sci_profession.html)

[CSAB 2000] *Computing Sciences Accreditation Board. Criteria for accrediting programs in computer science in the United States*. Version 1.0, January 2000. (http://www.csab.org/criteria2k_v10.html)

[CSTB 1994] *Computing Science and Telecommunications Board. Realizing the information future.* Washington DC: National Academy Press, 1994.

[CSTB 1999] *Computing Science and Telecommunications Board. Being fluent with information technology*. Washington DC: National Academy Press, 1999.

[Curtis 1983] Kent K. Curtis. *Computer manpower: Is there a crisis?* Washington DC: National Science Foundation, 1983. (http://www.acm.org/sigcse/papers/curtis83/)

[Cybulski 2000] J.L. Cybulski and T. Linden Learning *Systems Design with UML and Patterns*, IEEE Transactions on Education, vol 43, no 4, 372 – 376, November 2000

[Davis 1997] Gordon B. Davis, John T. Gorgone, J. Daniel Couger, David L. Feinstein, and Herbert E. Longnecker, Jr. *IS'97 model curriculum and guidelines for undergraduate degree programs in information systems*. Association of Information Technology Professionals, 1997. (http://webfoot.csom.umn.edu/faculty/gdavis/curcomre.pdf)

[Denning 1989] Peter J. Denning, Douglas E. Comer, David Gries, Michael C. Mulder, Allen B. Tucker, A. Joe Turner, and Paul R. Young. "Computing as a discipline", *Communications of the ACM*, 32(1):9-23, January 1989.

[Denning 1992] Peter J. Denning, Peter J., "Educating a New Engineer", *Communications of the ACM*, December, Vol. 35, No. 12, December 1992, pp. 83-97.

[Denning 1998] Peter J. Denning. *Computing the profession*. Educom Review, November 1998.

[Denning 1999] Peter J. Denning. *Our seed corn is growing in the commons*. Information Impacts Magazine, March 1999.

(http://www.cisp.org/imp/march_99/denning/03_99denning.htm)

[EAB 1983] Educational Activities Board. *The 1983 model program in computer science and engineering.* Technical Report 932, Computer Society of the IEEE, December 1983.

[EAB 1986] Educational Activities Board. *Design education in computer science and engineering*. Technical Report 971, Computer Society of the IEEE, October 1986.

[EC 1977] Education Committee of the IEEE Computer Society. *A curriculum in computer science and engineering.* Publication EHO119-8, Computer Society of the IEEE, January 1977.

[Fairley 1985] R. Fairley, *Software Engineering Concepts*, McGraw-Hill, 1985.

[Fellows 2002] Sharon Fellows, Richard Culver, Peter Ruggieri, William Benson *Instructional Tools for Promoting Self-directed Skills in Freshmen*, FIE 2002, Boston, November, 2002.

[Feisel 2002] Lyle D. Feisel, George D. Peterson, *Learning Objectives for Engineering Laboratories*, FIE 2002, Boston, November, 2002

[Fleddermann 2000] C.B. Fleddermann *Engineering Ethics Cases for Electrical and Computer Engineering Students*, IEEE Transactions on Education, vol 43, no 3, 284 – 287, August 2000.

[Ford 1994] Gary Ford, *A Progress Report on Undergraduate Software Engineering Education*, CMU/SEI-94-TR-11, Software Engineering Institute, Carnegie Mellon University, May 1994.

[Ford 1996] Gary Ford and Norman E. Gibbs, *A Mature Profession of Software Engineering*, CMU/SEI-96-TR-004, Software Engineering Institute, Carnegie Mellon University, January 1996.

[Gibbs 1986] Norman E. Gibbs and Allen B. Tucker "Model curriculum for a liberal arts degree in computer science", *Communications of the ACM*, 29(3):202-210, March 1986.

[Giladi 1999] R. Giladi, *An Undergraduate Degree Program for Communications Systems Engineering*, IEEE Transactions on Education, vol 42, no 4, 295 – 304, November 1999.

[Gorgone 2000] John T. Gorgone, Paul Gray, David L. Feinstein, George M. Kasper, Jerry N. Luftman, Edward A. Stohr, Joseph S. Valacich, and Rolf T. Wigand. MSIS 2000: Model curriculum and guidelines for graduate degree programs in information systems. Association for Computing Machinery and Association for Information Systems, January 2000. (http://cis.bentley.edu/ISA/pages/documents/msis2000jan00.pdf)

[Gorgone 2002]
John T. Gorgone, Gordon B. Davis, Joseph S valacich, Heikki Topi, David L. Feinstein, and Herbert E. Longenecker, Jr. *IS 2002: Model Curriculum for Undergraduate Degree Programs in Information Systems*, published by the ACM, 2002.

[Hilburn 2002a]
Thomas B. Hilburn, Software Engineering Education: A Modest Proposal, *IEEE Software*, Vol. 14, No. 4, November 1997.

[Hilburn, 2002b]
Thomas B. Hilburn and Watts S. Humphrey, The Impending Changes in Software Education, *IEEE Software*, Vol 19, No. 5, September / October, 22 – 24, 2002.

[Hunter 2001] Robin Hunter and Richard H. Thayer (editors) *Software Process Improvement,* published by the IEEE Computer Society, Los Alamitos, CA 2001

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

[IEEE 2001] Institute for Electrical and Electronic Engineers. *IEEE code of ethics.* Piscataway, NJ: IEEE, May 2001. (http://www.ieee.org/about/whatis/code.html)

[Kelemen 1999] Charles F. Kelemen (editor), Owen Astrachan, Doug Baldwin, Kim Bruce, Peter Henderson, Dale Skrien, Allen Tucker, and Charles Ban Loan. *Computer Science Report to the CUPM Curriculum Foundations Workshop in Physics and Computer Science.* Report from a workshop at Bowdoin College, October 28-31, 1999.

[Kemper 1990] J. Kemper, Engineers and Their Profession, Oxford University Press, 1990.

[Koffmanl 1984] Elliot P. Koffman, Philip L. Miller, and Caroline E. Wardle. *Recommended curriculum for CS1: 1984 a report of the ACM curriculum task force for CS1.* Communications of the ACM, 27(10):998-1001, October 1984.

[Koffman 1985] Elliot P. Koffman, David Stemple, and Caroline E. Wardle. Recommended curriculum for CS2, 1984: *A report of the ACM curriculum task force for CS2*, Communications of the ACM, 28(8):815-818, August 1985.

[Lee 1998] Edward A. Lee and David G. Messerschmitt. Engineering and education for the future, *IEEE Computer*, 77-85, January 1998.

[Lethbridge 2000] Timothy Lethbridge, T., What Knowledge is Important to a Software Engineer?, *IEEE Computer*, Vol 33, No. 6, pp. 44-50, May 2000.

[Lidtke 1999] Doris K. Lidtke, Gordon E. Stokes, Jimmie Haines, and Michael C. Mulder. ISCC '99: *An information systems-centric curriculum '99*, July 1999. (http://www.iscc.unomaha.edu)

[Lutz 2001] Michael J. Lutz *Software Engineering on Internet Time*, IEEE Computer, 34, 5, 36, May, 2001.

[Marciniak 1994] John Marciniak (editor-in-chief) *Encyclopedia of Software Engineering*, published by John Wiley & Sons Inc, New York 1994

[Martin 1996] C. Dianne Martin, Chuck Huff, Donald Gotterbarn, Keith Miller. *Implementing a tenth strand in the CS curriculum*. Communications of the ACM, 39(12):75-84, December 1996.

[McDermid, 1991] John McDermid (editor) *Software Engineer's Reference Book*, published by Butterworth-Heinemann Ltd, Oxford, England 1991

[Meyer 2001] Bertrand Meyer "Software Engineering in the Academy", *IEEE Computer*, 34,5, 28-35, May 2001.

[Mulder 1975] Michael C. Mulder. *Model curricula for four-year computer science and engineering programs: Bridging the tar pit.* Computer, 8(12):28-33, December 1975.

[Mulder 1984] Michael C. Mulder and John Dalphin. *Computer science program requirements and accreditation—an interim report of the ACM/IEEE Computer Society joint task force.* Communications of the ACM, 27(4):330-335, April 1984.

[Mulder 1998] Fred Mulder and Tom van Weert. *Informatics in higher education: Views on informatics and noninformatics curricula*. Proceedings of the IFIP/WG3.2 Working Conference on Informatics (computer science) as a discipline and in other disciplines: What is in common? London: Chapman and Hall, 1998.

[NACE 2003] National Association of Colleges and Employers. *Job Outlook 2003* . (http://www.naceweb.org/**)**

[Naur 1969] P. Naur and B. Randell (editors) *Software Engineering*: Report on a Conference Sponsored by the NATO Science Committee (7 – 11 October 1968). 1969.

[Neumann 1995] Peter G. Neumann. *Computer related risks*. New York: ACM Press, 1995.

[Nordheden and Hoeflich 1999] K.J. Nordheden and M.H. Hoeflich, Undergraduate Research and Intellectual Property Rights, *IEEE Software*, Vol 19 , No. 5, September / October, 22 – 24, 2002.Education, vol 42, no 4, 233 – 236, November 1999.

[NSF 1996] National Science Foundation Advisory Committee. *Shaping the future: New expectations for undergraduate education in science, mathematics, engineering, and technology*. Washington DC: National Science Foundation, 1996.

[NTIA 1999] National Telecommunications and Information Administration. *Falling through the Net: Defining the digital divide.* Washington, DC: Department of Commerce, November 1999.

[Nunamaker 1982] Jay F. Nunamaker, Jr., J. Daniel Couger, Gordon B. Davis. Information systems curriculum recommendations for the 80s: Undergraduate and graduate programs. Communications of the ACM, 25(11):781-805, November 1982.

[Oklobdzija 2002] Vojin G. Oklobdzija (editor) *The Computer Engineering Handbook*, published by CRC Press LLC, Florida, USA, 2002.

[OTA 1988] Office of Technology Assessment. *Educating scientists and engineers: Grade school to grad school.* OTA-SET-377. Washington, DC: U.S. Government Printing Office, June 1988.

[Paulk 1995] Mark Paulk, Bill Curtis, Mary Beth Chrissis, and Charles Weber. *The capability maturity model: Guidelines for improving the software process*. Reading, MA: Addison-Wesley, 1995.

[QAA 2000] Quality Assurance Agency for Higher Education. *A report on benchmark levels for computing*. Gloucester, England: Southgate House, 2000.

[PMI 2000] Project Management Institute, *Guide to the Project Management Body of Knowledge*, PMI, 2000.

[Ralston 1980] Anthony Ralston and Mary Shaw. *Curriculum '78—Is computer science really that unmathematical.* Communications of the ACM (23)2:67-70, February 1980.

[Ralston 2000] Anthony Ralston, Edwin D. Reilly, David Hemmendinger (editors) *Encyclopedia of Computer Science*, fourth edition, Nature Publishing Group, London, England, 2000

[Ramamoorthy 1996] C.V. Ramamoorthy and Wei-tek Thai, *Advances in Software Engineering*, Communications of the ACM, 29, 10, 47-58, October, 1996.

[Richard 1999] W. D. Richard, D. E. Taylor and D. M. Zar *A Capstone Computer Engineering Design Course,* IEEE Transactions on Education, vol 42, no 4, 288 – 294, November 1999.

[Roberts 2001] Eric Roberts and Gerald Engel (editors) *Computing Curricula 2001: Computer Science*, Report of The ACM and IEEE-Computer Society Joint Task Force on Computing Curricula, Final Report, December 15th, 2001.

[Roberts 1995] Eric Roberts, John Lilly, and Bryan Rollins. *Using undergraduates as teaching assistants in introductory programming courses: An update on the Stanford experience*. SIGCSE Bulletin (27)1:48-52, March 1995.

[Roberts 1999] Eric Roberts. *Conserving the seed corn: Reflections on the academic hiring crisis*. SIGCSE Bulletin (31)4:4-9, December 1999.

[Royce 1970] W. W. Royce, "Managing the Development of Large Software Systems: Concepts and Techniques," Proceedings, WESCON, August 1970.

[SAC 1967] President's Science Advisory Commission. *Computers in higher education.* Washington DC: The White House, February 1967.

[Saiedian 2002] Hossein Saiedian, Donald J. Bagert, and Nancy R. Mead *Software Engineering Programs: Dispelling the Myths and Misconceptions*, IEEE Software, vol 19 , no. 5, September / October, 35 – 41, 2002.

[Shaw 1985] Mary Shaw. *The Carnegie-Mellon curriculum for undergraduate computer science*. New York: Springer-Verlag, 1985.

[Shaw 1991] Mary Shaw and James E Tomayko. *Models for undergraduate courses in software engineering.* Pittsburgh: Software Engineering Institute, Carnegie Mellon University, January 1991.

[Shaw 1992] Mary Shaw. *We can teach software better*. Computing Research News 4(4):2-12, September 1992.

[Shaw 2002] Mary Shaw, "What makes good research in software engineering?", *International Journal on Software Tools for Technology Transfer*, vol 4, DOI 10.1007/s10009-002-0083-4, June 2002.

[Shaw 2001] Mary Shaw, "The Coming-of-Age of Software Architecture Research", *Proceedings of the 23rd International Conference on Software Engineering, Toronto*, pp. 656-664a, Canada, IEEE Computer Society, 2001.

[SIGCHI 1992] Special Interest Group on Computer-Human Interaction. ACM SIGCHI Curricula for Human-Computer Interaction. New York: Association for Computing Machinery, 1992.

[Thayer 1993] Richard H. Thayer and Andrew McGettrick (editors) *Software Engineering – a European Perspective*, published by the IEEE Computer Society Press, Los Alamitos, CA 1993

[Tremblay 2000] G. Tremblay *Formal Methods: Mathematics, Computer Science, or Software Engineering?* , IEEE Transactions on Education, vol 43, no 4, 377 – 382, November 2000.

[Tucker 1991] Allen B. Tucker, Bruce H. Barnes, Robert M. Aiken, Keith Barker, Kim B. Bruce, J. Thomas Cain, Susan E. Conry, Gerald L. Engel, Richard G. Epstein, Doris K. Lidtke, Michael C. Mulder, Jean B. Rogers, Eugene H. Spafford, and A. Joe Turner. *Computing Curricula '91.* Association for Computing Machinery and the Computer Society of the Institute of Electrical and Electronics Engineers, 1991.

[Umphress 2002] David A. Umphress, T. Dean Hendrix, and James H. Cross *Software Process in the Classroom: The Capstone Project Experience*, IEEE Software, vol 19 , no. 5, September / October, 78 – 85, 2002.

[Walker 1996] Henry M. Walker and G. Michael Schneider. *A revised model curriculum for a liberal arts degree in computer science.* Communications of the ACM, 39(12):85-95, December 1996.

[Zadeh 1968] Lofti A. Zadeh. *Computer science as a discipline.* Journal of Engineering Education, 58(8):913-916, April 1968.

Appendix A: Detailed Descriptions of Proposed Courses

For each of the numbered courses from Chapter 6 we provide a description of the anticipated coverage of SEEK provided by the course. In most cases coverage of SEEK is considerably less than the 40 lecture-equivalent-hours that we use as a benchmark for a 'complete' course. This leaves space for institutions and instructors to tailor the courses: Covering extra material, or covering the given material in more depth.

Important note: It is intended to expand this section to add learning objectives for each of the new courses defined here.

CCCS introductory courses

Since these courses are taken directly from CCCS, the reader should consult that volume for more details. Note that other CCCS courses could be substituted for these.

CS101$_I$ Programming Fundamentals

Total hours of SEEK coverage: 39
CMP.cf (30 core hours of 140) - Computer Science foundations
  CMP.cf.1 (13 core hours of 39) - Programming Fundamentals
  CMP.cf.2 (3 core hours of 31) - Algorithms, Data Structures/Representation
  CMP.cf.3 (2 core hours of 5) - Problem solving techniques
  CMP.cf.6 (1 core hour of 1) - Basic concept of a system
  CMP.cf.7 (1 core hour of 1) - Basic user human factors
  CMP.cf.8 (1 core hour of 1) - Basic developer human factors
  CMP.cf.9 (7 core hours of 12) - Programming language basics
  CMP.cf.10 (1 core hour of 10) - Operating system basics key concepts from CCCS
  CMP.cf.12 (1 core hour of 5) - Network communication basics
CMP.tl (1 core hour of 4) - Construction Tools
PRF.pr (4 core hours of 20) - Professionalism
  PRF.pr.2 - Codes of ethics and professional conduct
  PRF.pr.3 - Social, legal, historical, and professional issues and concerns
  PRF.pr.6 - The economic impact of software
MAA.rfd (1 core hour of 3) - Requirements fundamentals
DES.con (1 core hour of 3) - Software design concepts
  DES.con.1 - Definition of design
VAV.rev (1 core hour of 6) - Reviews
  VAV.rev.1 - Desk checking
VAV.tst (1 core hour of 21) - Testing
  VAV.tst.1 - Unit testing

CS102$_I$ The Object-Oriented Paradigm

Total hours of SEEK coverage: 36
CMP.cf (30 core hours of 140) - Computer Science foundations
  CMP.cf.1 (13 core hours of 39) - Programming Fundamentals
  CMP.cf.2 (3 core hours of 31) - Algorithms, Data Structures/Representation
  CMP.cf.3 (3 core hours of 5) - Problem solving techniques
  CMP.cf.4 (3 core hours of 5) - Abstraction -- use and support for
  CMP.cf.5 (2 core hours of 20) - Computer organization
  CMP.cf.9 (5 core hours of 12) - Programming language basics
  CMP.cf.11 (1 core hour of 10) - Database basics
CMP.ct (1 core hour of 20) - Construction technologies
  DES.con.4 - Design principles
DES.hci (3 core hours of 12) - Human computer interface design
  DES.hci.1 - General HCI design principles
VAV.fnd (1 core hour of 5) - V&V terminology and foundations
  VAV.fnd.1 - Objectives and constraints of V&V
EVO.pro (1 core hour of 6) - Evolution processes
  EVO.pro.1 - Basic concepts of evolution and maintenance

CS103 Data Structures and Algorithms

Total hours of SEEK coverage: 31
CMP.cf (30 core hours of 140) - Computer Science foundations
  CMP.cf.1 (13 core hours of 39) - Programming Fundamentals
  CMP.cf.2 (15 core hours of 31) - Algorithms, Data Structures/Representation
  CMP.cf.4 (2 core hours of 5) - Abstraction -- use and support for
  CMP.cf.9 - Programming language basics
VAV.tst (1 core hour of 21) - Testing
  VAV.tst.2 - Exception handling

Intermediate fundamental computer science courses

This is a sample of CCCS courses that can be used to teach required material in SEEK. Other combinations of CCCS courses could be used, or new courses could be created to cover the same material. If these three courses are used, then the result is to teach much material beyond the essentials in SEEK; however, that is never inappropriate.

CS220 Computer Architecture

Total hours of SEEK coverage: 15
CMP.cf (15 core hours of 140) - Computer Science foundations
  CMP.cf.5 (15 core hours of 20) - Computer organization

CS226 Operating Systems and Networking

Total hours of SEEK coverage: 16
CMP.cf (16 core hours of 140) - Computer Science foundations
  CMP.cf.2 (3 core hours of 31) - Algorithms, Data Structures/Representation
  CMP.cf.10 (9 core hours of 10) - Operating system basics key concepts from CCCS
  CMP.cf.12 (4 core hours of 5) - Network communication basics

CS270T Databases

Total hours of SEEK coverage: 13
CMP.cf (11 core hours of 140) - Computer Science foundations
  CMP.cf.2 (2 core hours of 31) - Algorithms, Data Structures/Representation
  CMP.cf.11 (9 core hours of 10) - Database basics
MAA.md (2 core hours of 19) - Modeling

Mathematics fundamentals courses

CS105 Discrete Structures I

Total hours of SEEK coverage: 24
CMP.cf (3 core hours of 140) - Computer Science foundations
  CMP.cf.5 (3 core hours of 20) - Computer organization
FND.mf (21 core hours of 56) - Mathematical foundations
  FND.mf.1 (6 core hours of 6) - Functions, Relations and Sets
  FND.mf.2 (5 core hours of 9) - Basic Logic
  FND.mf.3 (4 core hours of 9) - Proof Techniques
  FND.mf.4 (6 core hours of 6) - Basic Counting
  FND.mf.10 - Number Theory

CS106 Discrete Structures II

Total hours of SEEK coverage: 27
CMP.cf (5 core hours of 140) - Computer Science foundations
  CMP.cf.2 (5 core hours of 31) - Algorithms, Data Structures/Representation
FND.mf (19 core hours of 56) - Mathematical foundations
  FND.mf.2 (4 core hours of 9) - Basic Logic
  FND.mf.3 (5 core hours of 9) - Proof Techniques
  FND.mf.4 (core hours of 6) - Basic Counting
  FND.mf.5 (4 core hours of 5) - Graphs and Trees
  FND.mf.6 (6 core hours of 9) - Discrete Probability
MAA.md (3 core hours of 19) - Modeling

MA271-sta Statistics and Empirical Methods

Total hours of SEEK coverage: 18
FND.mf (3 core hours of 56) - Mathematical foundations
  FND.mf.6 (3 core hours of 9) - Discrete Probability
FND.ef (15 core hours of 23) - Engineering foundations for software
  FND.ef.1 - Empirical methods and experimental techniques
  FND.ef.2 - Statistical analysis

Nontechnical compulsory courses

In the following series of courses, total SEEK coverage totals far less than 40 hours, so additional material would be taught.

NT271-eco Engineering Economics

Total hours of SEEK coverage: 13
FND.ef (2 core hours of 23) - Engineering foundations for software
  FND.ef.5 - Engineering design
FND.ec (10 core hours of 10) - Engineering economics for software
MGT.pp (1 core hour of 6) - Project planning

NT181-com Group Dynamics and Communication

Total hours of SEEK coverage: 11
PRF.psy (3 core hours of 5) - Group dynamics / psychology
PRF.com (8 core hours of 10) - Communications skills
  MAA.rsd.1 - Requirements documentation basics

NT291-eth Professional Software Engineering Practice

Total hours of SEEK coverage: 14
PRF.pr (13 core hours of 20) - Professionalism
  PRF.pr.1 - Accreditation, certification, and licensing
  PRF.pr.2 - Codes of ethics and professional conduct
  PRF.pr.3 - Social, legal, historical, and professional issues and concerns
  PRF.pr.4 - The nature of, and role of professional societies
  PRF.pr.5 - The nature and role of software engineering standards
  PRF.pr.6 - The economic impact of software
QUA.cc (1 core hour of 2) - Software quality concepts and culture
  QUA.cc.2 - Society's concern for quality
  QUA.cc.3 - The costs and impacts of bad quality

SE+CS introductory courses - first year start

SE101 Introduction to software engineering and computing

Total hours of SEEK coverage: 35
CMP.cf (19 core hours of 140) - Computer Science foundations
  CMP.cf.1 (9 core hours of 39) - Programming Fundamentals
  CMP.cf.3 (2 core hours of 5) - Problem solving techniques
  CMP.cf.4 (1 core hour of 5) - Abstraction -- use and support for
  CMP.cf.5 (2 core hours of 20) - Computer organization
  CMP.cf.6 (1 core hour of 1) - Basic concept of a system
  CMP.cf.7 (1 core hour of 1) - Basic user human factors
  CMP.cf.8 (1 core hour of 1) - Basic developer human factors
  CMP.cf.9 (2 core hours of 12) - Programming language basics
CMP.ct (2 core hours of 20) - Construction technologies
CMP.tl (1 core hour of 4) - Construction Tools
FND.ef (2 core hours of 23) - Engineering foundations for software
  FND.ef.3 - Measuring individual's performance
  FND.ef.4 - Systems development
  FND.ef.5 - Engineering design
PRF.pr (2 core hours of 20) - Professionalism
MAA.tm (1 core hour of 12) - Types of models
MAA.rfd (2 core hours of 3) - Requirements fundamentals
MAA.er (1 core hour of 4) - Eliciting requirements
MAA.rsd (1 core hour of 6) - Requirements specification & documentation
DES.con (1 core hour of 3) - Software design concepts
DES.str (1 core hour of 6) - Software design strategies
DES.dd (1 core hour of 12) - Detailed design
VAV.tst (1 core hour of 21) - Testing

SE102 Software engineering and computing II

Total hours of SEEK coverage: 36
CMP.cf (23 core hours of 140) - Computer Science foundations
  CMP.cf.1 (12 core hours of 39) - Programming Fundamentals
  CMP.cf.3 (3 core hours of 5) - Problem solving techniques
  CMP.cf.4 (1 core hour of 5) - Abstraction -- use and support for
  CMP.cf.9 (4 core hours of 12) - Programming language basics
  CMP.cf.10 (1 core hour of 10) - Operating system basics key concepts from CCCS
  CMP.cf.11 (1 core hour of 10) - Database basics
  CMP.cf.12 (1 core hour of 5) - Network communication basics
PRF.pr (1 core hour of 20) - Professionalism
MAA.md (1 core hour of 19) - Modeling
MAA.rv (1 core hour of 3) - Requirements validation
DES.str (1 core hour of 6) - Software design strategies
DES.dd (1 core hour of 12) - Detailed design
DES.nst (1 core hours of 3) - Design notations and support tools
VAV.fnd (1 core hour of 5) - V&V terminology and foundations
VAV.rev (1 core hour of 6) - Reviews
VAV.tst (2 core hours of 21) - Testing
VAV.par (1 core hour of 4) - Problem analysis and reporting
EVO.pro (1 core hour of 6) - Evolution processes

Software engineering core courses

SE200 Software Engineering and computing III

Total hours of SEEK coverage: 38
CMP.cf (18 core hours of 140) - Computer Science foundations
  CMP.cf.1 (5 core hours of 39) - Programming Fundamentals
  CMP.cf.2 (6 core hours of 31) - Algorithms, Data Structures/Representation
  CMP.cf.4 (1 core hour of 5) - Abstraction -- use and support for
  CMP.cf.9 (6 core hours of 12) - Programming language basics
CMP.ct (3 core hours of 20) - Construction technologies
FND.ef (1 core hour of 23) - Engineering foundations for software
PRF.pr (2 core hours of 20) - Professionalism
MAA.md (1 core hour of 19) - Modeling
DES.con (2 core hours of 3) - Software design concepts
DES.str (1 core hour of 6) - Software design strategies
DES.ar (2 core hours of 9) - Architectural design
DES.hci (4 core hours of 12) - Human computer interface design
DES.ev (1 core hour of 3) - Design Evaluation
VAV.fnd (1 core hour of 5) - V&V terminology and foundations
VAV.rev (1 core hour of 6) - Reviews
PRO.imp (1 core hour of 10) - Process Implementation
MGT.con (1 core hour of 2) - Management concepts

SE201-int Introduction to Software Engineering for Software Engineers

Total hours of SEEK coverage: 34
CMP.ct (4 core hours of 20) - Construction technologies
  CMP.ct.1 - API design and use
  CMP.ct.2 - Code reuse and libraries
  CMP.ct.3 - Object-oriented run-time issues
FND.ef (3 core hours of 23) - Engineering foundations for software
  FND.ef.1 - Empirical methods and experimental techniques
  FND.ef.4 - Systems development
  FND.ef.5 - Engineering design
PRF.pr (1 core hour of 20) - Professionalism
MAA.md (2 core hours of 19) - Modeling
  MAA.md.1 - Modelling principles
  MAA.md.2 - Pre & post conditions, invariants
  MAA.md.3 - Introduction to mathematical models and specification languages
MAA.tm (1 core hour of 12) - Types of models
MAA.rfd (1 core hour of 3) - Requirements fundamentals
MAA.er (1 core hour of 4) - Eliciting requirements
MAA.rsd (1 core hour of 6) - Requirements specification & documentation
  MAA.rsd.3 - Specification languages
MAA.rv (1 core hour of 3) - Requirements validation
DES.con (2 core hours of 3) - Software design concepts
DES.str (3 core hours of 6) - Software design strategies
DES.ar (2 core hours of 9) - Architectural design
DES.hci (1 core hour of 12) - Human computer interface design
DES.dd (2 core hours of 12) - Detailed design
DES.nst (1 core hour of 3) - Design notations and support tools
DES.ev (1 core hour of 3) - Design Evaluation
VAV.fnd (1 core hour of 5) - V&V terminology and foundations
VAV.rev (1 core hour of 6) - Reviews
VAV.tst (2 core hours of 21) - Testing
VAV.par (1 core hour of 4) - Problem analysis and reporting
PRO.imp (1 core hour of 10) - Process Implementation
MGT.con (1 core hour of 2) - Management concepts

SE211-con Software Construction

Total hours of SEEK coverage: 36
CMP.ct (10 core hours of 20) - Construction technologies
 CMP.ct.6 - Error handling, exception handling, and fault tolerance
 CMP.ct.7 - State-based and table driven construction techniques
 CMP.ct.8 - Run-time configuration and internationalization
 CMP.ct.9 - Grammar-based input processing
 CMP.ct.10 - Concurrency primitives
 CMP.ct.11 - Middleware
 CMP.ct.12 - Construction methods for distributed software
 CMP.ct.14 - Hot-spot analysis and performance tuning
CMP.tl (3 core hours of 4) - Construction Tools
CMP.fm (8 core hours of 8) - Formal construction methods
FND.mf (11 core hours of 56) - Mathematical foundations
 FND.mf.5 (1 core hour of 5) - Graphs and Trees
 FND.mf.7 (4 core hours of 4) - Finite State Machines, regular expressions
 FND.mf.8 (4 core hours of 4) - Grammars
 FND.mf.9 (2 core hours of 4) - Numerical precision, accuracy and errors
MAA.md (4 core hours of 19) - Modeling

SE212-hci Software Engineering Approach to Human Computer Interaction

Total hours of SEEK coverage: 25
CMP.ct (1 core hour of 20) - Construction technologies
 CMP.ct.8 - Run-time configuration and internationalization
 CMP.tl.2 - GUI builders
FND.ef (3 core hours of 23) - Engineering foundations for software
PRF.psy (1 core hour of 5) - Group dynamics / psychology
MAA.md (4 core hours of 19) - Modeling
MAA.tm (1 core hour of 12) - Types of models
 MAA.rfd.5 - Analyzing quality
DES.hci (6 core hours of 12) - Human computer interface design
VAV.fnd (1 core hour of 5) - V&V terminology and foundations
 VAV.fnd.4 - Metrics & Measurement
VAV.rev (1 core hour of 6) - Reviews
 VAV.rev.3 - Inspections
 VAV.tst.9 - Testing across quality attributes
VAV.hct (6 core hours of 6) - Human computer user interface testing and evaluation
QUA.pda (1 core hour of 4) - Product assurance
 QUA.pda.6 - Assessment of product quality attributes

SE213-hld Design and Architecture of Large Software Systems

Total hours of SEEK coverage: 28
MAA.md (5 core hours of 19) - Modeling
MAA.tm (5 core hours of 12) - Types of models
DES.str (2 core hours of 6) - Software design strategies
DES.ar (5 core hours of 9) - Architectural design
  VAV.tst.1 - Unit testing
EVO.pro (3 core hours of 6) - Evolution processes
  EVO.pro.1 - Basic concepts of evolution and maintenance
  EVO.pro.2 - Relationship between evolving entities
EVO.ac (2 core hours of 4) - Evolution Activities
MGT.con (1 core hour of 2) - Management concepts
MGT.pp (1 core hour of 6) - Project planning
MGT.cm (4 core hours of 5) - Software configuration management

SE221-tes Testing

Total hours of SEEK coverage: 23
MAA.rfd (1 core hour of 3) - Requirements fundamentals
  MAA.rfd.4 - Requirements characteristics
VAV.fnd (2 core hours of 5) - V&V terminology and foundations
VAV.rev (1 core hour of 6) - Reviews
VAV.tst (14 core hours of 21) - Testing
  VAV.tst.2 - Exception handling
VAV.par (3 core hours of 4) - Problem analysis and reporting
QUA.pda (2 core hours of 4) - Product assurance

SE311-des Software Design and Evolution

Total hours of SEEK coverage: 33
CMP.ct (3 core hours of 20) - Construction technologies
  CMP.ct.11 - Middleware
  CMP.ct.12 - Construction methods for distributed software
  CMP.ct.13 - Constructing heterogeneous systems
MAA.md (4 core hours of 19) - Modeling
  MAA.tm.3 - Structure modelling
DES.str (2 core hours of 6) - Software design strategies
DES.ar (5 core hours of 9) - Architectural design
DES.dd (8 core hours of 12) - Detailed design
DES.nst (1 core hour of 3) - Design notations and support tools
DES.ev (1 core hour of 3) - Design Evaluation
EVO.pro (5 core hours of 6) - Evolution processes
EVO.ac (4 core hours of 4) - Evolution Activities

SE312-lld Low-Level Design

Total hours of SEEK coverage: 26
CMP.ct (13 core hours of 20) - Construction technologies
CMP.tl (3 core hours of 4) - Construction Tools
CMP.fm (2 core hours of 8) - Formal construction methods
MAA.tm (2 core hours of 12) - Types of models
DES.dd (5 core hours of 12) - Detailed design
  VAV.tst.6 - Developing test cases based on use cases and/or customer stories
EVO.ac (1 core hour of 4) - Evolution Activities

SE321-qvv Quality, verification and validation

Total hours of SEEK coverage: 37
FND.mf (2 core hours of 56) - Mathematical foundations
  FND.mf.9 (2 core hours of 4) - Numerical precision, accuracy and errors
VAV.fnd (2 core hours of 5) - V&V terminology and foundations
VAV.rev (1 core hour of 6) - Reviews
VAV.tst (14 core hours of 21) - Testing
VAV.par (3 core hours of 4) - Problem analysis and reporting
PRO.con (1 core hour of 3) - Process concepts
QUA.cc (1 core hour of 2) - Software quality concepts and culture
QUA.std (2 core hours of 2) - Software quality standards
QUA.pro (4 core hours of 4) - Software quality processes
QUA.pca (4 core hours of 4) - Process assurance
QUA.pda (3 core hours of 4) - Product assurance

SE322-req Requirements

Total hours of SEEK coverage: 18
MAA.tm (9 core hours of 12) - Types of models
MAA.rfd (1 core hour of 3) - Requirements fundamentals
MAA.er (2 core hours of 4) - Eliciting requirements
MAA.rsd (4 core hours of 6) - Requirements specification & documentation
MAA.rv (1 core hour of 3) - Requirements validation
MAA.mgt (1 core hour of 3) - Requirements management

SE323-pmt Project Management

Total hours of SEEK coverage: 26
MAA.mgt (2 core hours of 3) - Requirements management
PRO.con (2 core hours of 3) - Process concepts
PRO.imp (9 core hours of 10) - Process Implementation
MGT.con (1 core hour of 2) - Management concepts
MGT.pp (3 core hours of 6) - Project planning
MGT.per (1 core hour of 2) - Project personnel and organization
MGT.ctl (4 core hours of 4) - Project control
MGT.cm (4 core hours of 5) - Software configuration management

SE324-pro Process and Management

Total hours of SEEK coverage: 39
MAA.er (2 core hours of 4) - Eliciting requirements
MAA.rsd (1 core hour of 6) - Requirements specification & documentation
MAA.mgt (3 core hours of 3) - Requirements management
  VAV.tst.5 - Integration Testing
EVO.pro (2 core hours of 6) - Evolution processes
  EVO.pro.3 - Models of software evolution
  EVO.pro.4 - Cost models of evolution
PRO.con (3 core hours of 3) - Process concepts
PRO.imp (9 core hours of 10) - Process Implementation
QUA.cc (1 core hour of 2) - Software quality concepts and culture
QUA.std (2 core hours of 2) - Software quality standards
QUA.pro (4 core hours of 4) - Software quality processes
QUA.pca (4 core hours of 4) - Process assurance
QUA.pda (1 core hour of 4) - Product assurance
MGT.pp (2 core hours of 6) - Project planning
MGT.per (1 core hour of 2) - Project personnel and organization
MGT.ctl (4 core hours of 4) - Project control

SE313-fm Formal Methods in Software Engineering

Total hours of SEEK coverage: 34
CMP.fm (6 core hours of 8) - Formal construction methods
FND.mf (13 core hours of 56) - Mathematical foundations
  FND.mf.5 (1 core hour of 5) - Graphs and Trees
  FND.mf.7 (4 core hours of 4) - Finite State Machines, regular expressions
  FND.mf.8 (4 core hours of 4) - Grammars
  FND.mf.9 (4 core hours of 4) - Numerical precision, accuracy and errors
MAA.md (3 core hours of 19) - Modeling
  MAA.md.3 - Introduction to mathematical models and specification languages
MAA.tm (2 core hours of 12) - Types of models
  MAA.tm.2 - Behavioral modelling
MAA.rsd (3 core hours of 6) - Requirements specification & documentation
  MAA.rsd.3 - Specification languages
MAA.rv (1 core hour of 3) - Requirements validation
DES.dd (3 core hours of 12) - Detailed design
DES.nst (1 core hour of 3) - Design notations and support tools
  DES.nst.6 - Formal design analysis
DES.ev (1 core hour of 3) - Design Evaluation
  DES.ev.2 - Evaluation techniques
EVO.ac (1 core hour of 4) - Evolution Activities
  EVO.ac.6 - Refactoring
  EVO.ac.7 - Program transformation

Capstone project course

SE400-cap Software Engineering Capstone Project

This material represents SEEK units that must be practiced in all projects. Beyond this, different projects will exercise skills in different areas of SEEK.

Total hours of SEEK coverage: 28
CMP.ct (1 core hour of 20) - Construction technologies
PRF.psy (1 core hour of 5) - Group dynamics / psychology
PRF.com (2 core hours of 10) - Communications skills
PRF.pr (2 core hours of 20) - Professionalism
MAA.tm (1 core hour of 12) - Types of models
MAA.er (1 core hour of 4) - Eliciting requirements
MAA.rsd (1 core hour of 6) - Requirements specification & documentation
MAA.rv (1 core hour of 3) - Requirements validation
DES.str (1 core hour of 6) - Software design strategies
DES.ar (2 core hours of 9) - Architectural design
DES.hci (2 core hours of 12) - Human computer interface design
DES.dd (2 core hours of 12) - Detailed design
DES.nst (1 core hour of 3) - Design notations and support tools
DES.ev (1 core hour of 3) - Design Evaluation
VAV.rev (2 core hours of 6) - Reviews
VAV.tst (3 core hours of 21) - Testing
MGT.pp (2 core hours of 6) - Project planning
MGT.per (1 core hour of 2) - Project personnel and organization
MGT.cm (1 core hour of 5) - Software configuration management

Appendix B: Skills and exercises

Software engineering curricula must not only teach facts, they must also ensure that students achieve a level of skill at doing particular tasks required of the practicing software engineer. This means that students must learn by doing exercises that will enable them to build up the requisite level of skill. Most of the exercises will be problem-solving in nature. Therefore, in this section, we list a minimal set of types of exercises that should be part of the education of all software engineering undergraduates.

We primarily consider exercises for SEEK topics that have a Bloom's taxonomy category of 'a' (application). Some of the exercises may also help students master material in the 'c' (comprehension) or 'k' (knowledge) categories; however, simple reading or lectures may suffice for many of these.

The process of developing this section

The first pass at writing this section consisted of looking at each SEEK topic given a Bloom's taxonomy category of 'a' (application), and describing the types of exercises to achieve application-level mastery of that topic. The result, however, was a massive list that could not possibly be tackled in a four-year software engineering degree. What we provide below, therefore, is a shorter list in which many of the exercises can be used to help master several of the KAs.

The list of exercise categories

The following table specifies exercise categories very broadly, leaving an opportunity for instructors and textbook authors to be far more specific. In most cases, students would be expected to do exercises in each category many times, each time deepening their skills and learning about new tools, methods, technologies or domains.

Exercise Category	Relevant SEEK units/topics	Relevant courses
Exercise categories primarily oriented towards the CMP knowledge area		
Write algorithms for a variety of problems in several different domains.		
Analyze the computational complexity of several different algorithms.		
Implement carefully documented small programs or changes to larger programs, where the programs are written in several different programming languages, and where the power and capabilities of the languages are effectively exploited.		
Find and correct defects in systems of a variety of types and size		
Perform desk-checking or inspection of programs, and record the results		
Build systems or subsystems that interact with other well-specified systems or subsystems.		
Choose appropriate algorithms, data structures, API calls and reusable libraries for a variety of problems.		

Exercise Category	Relevant SEEK units/topics	Relevant courses
Given a variety of desired attributes, choose among several candidate implementations.		
Build systems involving middleware		
Build a distributed system		
Build a system involving parsing technology		
Measure and analyze the performance a variety of systems		
Understand a small system, and analyze the effect of changes.		
Exercise categories primarily oriented towards the FND knowledge area		
Apply methods from mathematical logic to the analysis of complex conditions.		
Write small proofs of program correctness		
Write small formal specifications for a variety of types of problems		
Write constraints of various kinds in different types of system model		
Find mistakes and errors in logic in a variety of system model		
Perform statistical analysis of experimental results		
Exercise categories primarily oriented towards the MAA knowledge area		
Create class diagrams of a variety of domains		
Create class diagrams of a variety of systems		
Create state diagrams and other behavioural models of a variety of systems		
Elicit requirements for a variety of problems.		
Write good quality requirements documents		
Exercise categories primarily oriented towards the DES knowledge area		
Write well reasoned descriptions of the design of a variety of small systems or features, following one or more published design methods		
Analyze the effects of a variety of design decisions		
Exercise categories primarily oriented towards the VAV knowledge area		
Perform a code inspection		
Write test cases for a variety of types of software		
Test a variety of types of software according to an established test plan		
Perform heuristic evaluation and user testing of a user interface		
Exercise categories primarily oriented towards the MGT, QUA and PRO knowledge areas		
Write aspects of project plans for a variety of types of projects		
Write a quality plan		
Use Gantt and Pert charts to develop schedules for a software project		
Estimate the costs of a variety of software engineering activities		
Track changes to code and other documents using a configuration management tool		
Exercise categories primarily oriented towards the PRF		

Exercise Category	Relevant SEEK units/topics	Relevant courses
knowledge area		
Work in teams on many of the activities described above		

Appendix C: Contributors and Reviewers

Education Knowledge Area Volunteers

Jonathan D. Addelston, UpStart Systems, U.S.
Roger Alexander, Colorado State University, U.S.
Niniek Angkasaputra, Fraunhofer Institute of Experimental Software Engineering, Germany
Mark A. Ardis, Rose-Hulman University, U.S.
Jocelyn Armarego, Murdoch University, Australia
Doug Baldwin, The State University of New York, Geneseo, U.S.
Earl Beede, Construx, U.S.
Fawsy Bendeck, University of Kaiserslautern, Germany
Mordechai Ben-Menachem, Ben-Gurion University, Israel
Robert Burnett, consultant, Brazil
Kai Chang, Auburn University, U.S.
Jason Chen, National Central University, Taiwan
Cynthia Cicalese, Marymount University, U.S.
Tony (Anthony) Cowling, University of Sheffield, U.K.
David Dampier, Mississippi State University, U.S.
Mel Damodaran, University of Houston, U.S.
Onur Demirors, Middle East Technical University, Turkey
Vladan Devedzic, University of Belgrade, Yugoslavia
Oscar Dieste, University of Alfonso X El Sabio, Spain
Dick Fairley Oregon Graduate Institute, U.S.
Mohamed E. Fayad, University of Nebraska, Lincoln, U.S.
Orit Hazzan, Israel Institute of Technology, Israel
Bill Hefley, consultant, U.S.
Peter Henderson, Butler University, U.S.
Joel Henry, University of Montana, U.S.
Jens Jahnke, University of Victoria, Canada
Stanislaw Jarzabek, National University of Singapore, Singapore
Natalia Juristo, Universidad Politecnica of Madrid, Spain
Umit Karakas, consultant, Turkey
Atchutarao Killamsetty, JENS SpinNet, Japan
Haim Kilov, Financial Systems Architects, U.S.
Moshe Krieger, University of Ottawa, Canada
Hareton Leung, Hong Kong Polytechnic University, Hong Kong
Marta Lopez, Fraunhofer Institute of Experimental Software Engineering, Germany
Mike Lutz, Rochester Institute of Technology, U.S.
Paul E. MacNeil, Mercer University, U.S.
Mike McCracken, Georgia Institute of Technology, U.S.
James McDonald, Monmouth University, U.S.
Emilia Mendes, University of Auckland, New Zealand
Luisa Mich, University of Trento, Italy
Ana Moreno, Universidad Politecnica of Madrid, Spain
Traian Muntean, University of Marseilles, France

Keith Olson, Utah Valley State College, U.S.
Michael Oudshoorn, University of Adelaide, Australia
Dietmar Pfahl, Fraunhofer Institute of Experimental Software Engineering, Germany
Mario Piattini, University of Paseo, Spain
Francis Pinheiro, University of Brazil, Brazil
Valentina Plekhanova, University of Sunderland, U.K.
Hossein Saiedian, University of Kansas, U.S.
Stephen C. Schwarm, EMC, U.S.
Peraphon Sophatsathit, Chulalongkorn University, Thailand
Jennifer S. Stuart, Construx, U.S.
Linda T. Taylor, Taylor & Zeno Systems, U.S.
Richard Thayer, California State University, Sacramento, U.S.
Jim Tomayko, Carnegie Melon University, U.S.
Massood Towhidnejad, Embry-Riddle University, U.S.
Joseph E. Urban, Arizona State University, U.S.
Arie van Deursen, National Research Institute for Mathematics & Computer Science, Netherlands
Sira Vegas, University of Madrid, Spain
Bimlesh Wadhwa, National University of Singapore, Singapore
Yingxu Wang, University of Calgary, Canada
Mary Jane Willshire, University of Portland, U.S.
Mansour Zand, University of Nebraska, Omaha, U.S.
Jianhan Zhu, University of Ulster, U.K.

CCSE SEEK Workshop Attendees

Earl Beede, Construx, U.S.
Pierre Bourque, University of Quebec
David Budgen, Keele University, U.K.
Kai Chang, Auburn University, U.S.
Jorge L. Díaz-Herrera, Rochester Institute of Technology, U.S.
Frank Driscoll, Mitre Cooperation, U.S.
Steve Easterbrook, University of Toronto, Canada
Dick Fairley, Oregon Graduate Institute, U.S.
Peter Henderson, Butler University, U.S.
Thomas B. Hilburn, Embry-Riddle University, U.S.
Tom Horton, University of Virginia, U.S.
Cem Kaner, Florida Institute of Technology, U.S.
Haim Kilov, Financial Systems Architects, U.S.
Gideon Kornblum, Getronics, Netherlands
Rich LeBlanc, Georgia Institute of Technology, U.S.
Timothy C. Lethbridge, University of Ottawa, Canada
Bill Marion, Valparaiso University, U.S.
Yoshihiro Matsumoto, Musashi Institute of Technology, Japan
Mike McCracken, Georgia Institute of Technology, U.S.
Andrew McGettrick, University of Strathclyde, U.K.
Susan Mengel, Texas Tech University, U.S.

Traian Muntean, University of Marseilles, France
Keith Olson, Utah Valley State College, U.S.
Allen Parrish, University of Alabama, U.S.
Ann Sobel, Miami University, U.S.
Jenny Stuart, Construx, U.S.
Linda T. Taylor, Taylor & Zeno Systems, U.S.
Barrie Thompson, University of Sunderland, U.K.
Richard Upchurch, University of Massachussetts, U.S.
Frank H. Young, Rose-Hulman University, U.S.

SEEK Internal Reviewers

Barry Boehm, University of Southern California, U.S.
Kai H. Chang, Auburn University, U.S.
Jason Jen-Yen Chen, National Central University, Taiwan
Tony Cowling, University of Sheffield, U.K.
Vladan Devedzic, University of Belgrade, Yugoslavia
Laura Dillon, Michigan State University, U.S.
Dennis J. Frailey, Raytheon, U.S.
Peter Henderson, Butler University, U.S.
Watts Humphrey, Software Engineering Institute, U.S.
Haim Kilov, Financial Systems Architects, U.S.
Hareton Leung, Hong Kong Polytechnic University, Hong Kong
Yoshihiro Matsumoto, Information Processing Society, Japan
Bertrand Meyer, ETH, Zurich
Luisa Mich, University of Trento, Italy
James W. Moore, Mitre, U.S.
Hausi Muller, University of Victoria, Canada
Peter G. Neuman, SRI International, U.S.
David Notkin, University of Washington, U.S.
David Parnas, McMaster University, Canada
Dietmar Pfahl, Fraunhofer Institute of Experimental Software Engineering, Germany
Mary Shaw, Carnegie Mellon University, U.S.
Ian Sommerville, Lancaster University, U.K.
Peraphon Sophatsathit, Chulalongkorn University, Thailand
Steve Tockey, Construx Software, U.S.
Massood Towhidnejad, Embry-Riddle University, U.S.
Leonard Tripp, Boeing Shared Services, U.S.

SEEK External Reviewers

James P. Alstad, Hughes Space and Communications Company, USA
Niniek Angkasaputra, Fraunhofer Institute for Experimental SE, Germany
Hernan Astudillo, Financial Systems Architects, USA
Donald J. Bagert, Rose-Hulman Institute of Technology, USA
Mario R. Barbacci, Software Engineering Institute, USA
Ilia Bider, IbisSoft AB, Sweden

Grady Booch, Rational Corp, USA
Jurgen Borstler, Umeå University, Sweden
Pierre Bourque, Ecole de Technologie Superieure, Montreal, Canada
David Budgen, Keele University, UK
Joe Clifton, University of Wisconsin - Platteville, USA
Kendra Cooper, The University of Texas at Dallas, USA
Tony Cowling, University of Sheffield, UK
Vladan Devedzic, University of Belgrade, Yogoslavia
Rick Duley, Edith Cowan University, Australia
Robert Dupuis, Universite de Quebec à Monteal, Canada
Juan Garbajosa, Universidad Politecnica de Madrid, Spain
Robert L. Glass, Indiana University, USA
Orit Hazzan, Technion -- Israel Institute of Technology, Israel
Hui Huang, National Institute of Standards and Technology, USA
IFIP Working Group 2.9
Joseph Kasser, University of South Australia
Khaled Khan, University of Western Sydney, Australia
Peter Knoke, University of Alaska, Fairbanks, USA
Gideon Kornblum, CManagement bv, Netherlands
Claude Laporte, Ecole de Technologie Superieure, Montreal, Canada
Ansik Lee, Texas Instruments, USA
Hareton Leung, Hong Kong Polytechnic University, Hong Kong
Grace Lewis, Software Engineering Institute, USA
Michael Lutz, Rochester Institute of Technology, USA
Andrew Malton, University of Waterloo, Canada
Nikolai Mansurov, KLOCwork Inc., Ottawa, Canada
Esperanza Marcos, Rey Juan Carlos University, Spain
Pat Martin, Florida Institute of Technology, USA
Kenneth L. Modesitt, Indiana University - Purdue University Fort Wayne, USA
Ibrahim Mohamed, Universiti Kebangsaan, Malaysia
James Moore, Mitre Corporation, USA
Keith Paton, Independent consultant, Montreal, Canada
Pedagogy Focus Group Volunteers
Valentina Plekhanova, University of Sunderland, UK
Steve Roach, University of Texas at El Paso, USA
Francois Robert, Ecole de Technologie Superieure, Montreal, Canada
Robert C. Seacord, Software Engineering Institute, USA
Peraphon Sophatsathit, Chulalongkorn University, Thailand
Witold Suryn, Ecole de Technologie Superieure, Montreal, Canada
Sylvie Trudel, Ecole de Technologie Superieure, Montreal, Canada
Hans van Vliet, Vrije Universiteit Amsterdam, Netherlands
Frank H. Young, Rose-Hulman Institute of Technology, USA
Zdzislaw Zurakowski, Institute of Power Systems Automation, Poland

CCSE Pedagogy volunteers:

Jonathan Addelston, USA

Donald Bagert, Rose-Hulman Institute of Technology, USA
Jürgen Börstler, Umea Universitet, Sweden
David Budgen, Keele University, United Kingdom
Joe Clifton, University of Wisconsin, Plattsburgh, USA
Kendra Cooper, University of Texas, Dallas, USA
Vladan Devedzic, University of Belgrade, Yugoslavia
Rick Duley, Perth, Western Australia
Garth Glynn, University of Brighton, UK
Elizabeth Hawthorne, Union County College, USA
Orit Hazzan, Technion, Israel
Justo Hidalgo, Universidad Antonio de Nebrija, Spain
M. Umit Karakas, Turkey
Khaled Khan, University of Western Sydney, Australia
Yoshihiro Matsumoto, ASTEM Research Institute of Kyoto, Japan
Pat McGee, Florida Institute of Technology
Andrew McGettrick, University of Strathclyde, USA
Bruce Maxim, University of Michigan, USA
Ken Modesitt, Indiana University, USA
Steve Roach, University of Texas at El Paso, USA
Anthony Ruocco, Roger Williams University, USA
Peraphon Sophatsathit, Chulalongkorn University, Thailand
Barrie Thompson, University of Sunderland, UK
Yingxu Wang, University of Calgary, Canada
Frank H. Young, Rose-Hulman Institute of Technology, USA

Additional volunteers that participated in reviews of subsequent CCSE drafts will be added later.