The ACM Two-Year College Education Committee

Robert D. Campbell, Rock Valley College
Committee Chair

Elizabeth K. Hawthorne, Union County College

Karl J. Klee, Alfred State College

The ACM Two-Year College Education Committee gratefully acknowledges the outstanding contributions to the development of this report provided by our colleagues, including Peter Drexel, Plymouth State University; Becky Grasser, Lakeland Community College; Norma E. Hall, Manor College; John Impagliazzo, Hofstra University; Andrew McGettrick, University of Strathclyde, United Kingdom; Eric Roberts, Stanford University, as well as the previous work by the joint ACM/IEEE-CS Software Engineering Task Force in the development of the undergraduate report *Software Engineering 2004: Curriculum Guidelines for Undergraduate Degree Programs in Software Engineering*.

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

Computing Curricula 2005: Guidelines for Associate-Degree Transfer Curriculum in Software Engineering

The ACM Two-Year College Education Committee
and
The Joint Task Force on Software Engineering
Association for Computing Machinery
IEEE Computer Society

Table of Contents

List of Tables

Section 1: The Goal of This Report

This report provides guidelines for a software engineering curriculum track within the computer science degree program at associate-degree granting institutions. The report focuses on a program of study designed for students intending to transfer into baccalaureate programs awarding software engineering degrees. This report is specifically designed to promote articulation by linking software engineering curriculum in two-year colleges with that in baccalaureate institutions.

There are three major recent curriculum reports that provide foundation for this work.

- Computer Science curricula guidelines for undergraduate programs were finalized and approved in 2001, and were published under the title *Computing Curricula 2001: Computer Science.* This work was the result of the Joint Task Force on Computing Curricula 2001 established by the Institute of Electrical and Electronics Engineers Computer Society (IEEE-CS) and the Association for Computing Machinery (ACM). That report, together with accompanying materials, can be found at puter.org/education/.

- Computer Science curricula guidelines for associate-degree granting institutions were finalized and approved in 2003, and were published under the title *Computing Curricula 2003: Guidelines for Associate-Degree Curricula in Computer Science.* This work was the result of the IEEE-CS/ACM Joint Task Force on Computing Curricula 2001 and the ACM Two-Year College Education Committee. That report, together with accompanying materials, can be found at http://www.acmtyc.org/.

  The body of knowledge for associate-degree Computer Science is defined by the following areas: Algorithms and Complexity, Architecture and Organization, Discrete Structures, Graphics and Visual Computing, Human-Computer Interaction, Information Management, Net-Centric Computing, Operating Systems, Programming Fundamentals, Programming Languages, Software Engineering, and Social and Professional Issues.

- Software Engineering curricula guidelines for undergraduate programs were finalized and approved in 2004, and were published under the title *Software Engineering 2004: Curriculum Guidelines for Undergraduate Degree Programs in Software Engineering.* This work was the result of a joint task force of the ACM and IEEE-CS. That report, together with accompanying materials, can be found at puter.org/education/.

This report, *Computing Curricula 2005: Guidelines for Associate-Degree Transfer Curriculum in Software Engineering*, shares common goals and outcomes with the three above-mentioned curriculum reports. In the United States, as many as one-half of baccalaureate graduates initiate their studies in associate-degree granting institutions. For this reason, it is important to outline a software engineering curriculum track that can be initiated in the two-year college setting, specifically designed for seamless transfer into an upper-division program. This report recommends a program of study that specifically fulfills this requirement. However, it must be noted that the aims and objectives for software engineering undergraduate degree programs can vary from one institution to another for a variety of reasons. Ultimately, students are best served when institutions establish well defined articulation agreements between associate-degree and undergraduate-degree programs.

It is critical to note that 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 upper division level. For some students, this may require more than two years of study at the associate level. Particular attention must be paid to matching individual students to appropriate programs of study, taking into account each student's career goals and aspirations, talents and abilities, and life constraints such as time, finances, and geography.

By basing this report on three recently published sets of international curricula guidelines, the following goals are fulfilled:

- The use of computer science and mathematics courses from the *Computing Curricula 2003: Guidelines for Associate-Degree Curricula in Computer Science* report enables two-year colleges in the United States to incorporate a software engineering track easily into an existing computer science transfer degree program, irrespective of the specific department offering the degree.
- The incorporation of the software engineering philosophy, concepts, coursework and outcomes from the *Software Engineering 2004: Curriculum Guidelines for Undergraduate Degree Programs in Software Engineering* report helps to properly prepare students and facilitates seamless articulation.
- The report can be used to implement an introductory software engineering curriculum in countries outside the United States whose institutions have missions consistent with the US two-year college model. Using the *Computing Curricula 2003: Guidelines for Associate-Degree Curricula in Computer Science* report as a roadmap, students pursuing computer science could easily prepare for studies in software engineering should they decide to alter their career plans.

Section 2: The Nature of Software Engineering

Software engineering is more than just coding – it involves creating high-quality reliable products in a systematic, controlled, and efficient manner, with important emphases on analysis and evaluation, specification, design, and evolution. Many software products are among the most complex of man-made systems, requiring programming techniques and processes that scale well to the development of large applications, and that address the ongoing demand for new and evolved software, all within acceptable timeframes and budgets. For these reasons, software engineering requires both the analytical and descriptive tools developed in computer science and the rigor that the engineering disciplines bring to the reliability and trustworthiness of the artifacts that software producers design and develop.

In particular, the field of software engineering:

- Must be viewed as a discipline with stronger ties to computer science than it has to other engineering fields.
- Must share common characteristics with other engineering disciplines, including quantitative measurement, structured decision making, effective use of tools, and artifact reuse.
- Must apply engineering methods and practices to the development of software, with special emphasis on the development of large software systems.
- Must integrate the principles of discrete mathematics and computer science with engineering methodologies.
- Must utilize ion and modeling, and effective change management.
- Must include the quality control concepts of manufacturing process design.
- Must emphasize communication skills, teamwork skills, and professional principles and best practices.

Given then that software engineering is built upon the foundations of both computer science and engineering, the software engineering curriculum can be approached from either a *computer science-first* or *software engineering-first* perspective. There is clearly merit to each approach, and indeed the *Software Engineering 2004: Curriculum Guidelines for Undergraduate Degree Programs in Software Engineering* report provides two distinct introductory course sequences ("CS-first" and "SE-first") that in the end deliver students to the same point of preparation for more advanced study in the upper division.

While some suggest that the engineering-first approach better ensures that students develop a proper sense of the field in the context of engineering, the computer science-first approach is much more prevalent, and for many reasons likely to remain so. This report is based on the computer science-first approach for the following reasons:

- Students with limited programming experience may not have the necessary background or context for the study of software engineering concepts in their introductory courses.
- The current guidelines for foundation computer science curricula, which have greatly influenced the coursework now in place at many institutions, include concepts and programming paradigms that must be mastered through study and

practice. Once in place, these skills can be honed and refined in subsequent coursework, including the study of other software engineering topics.

- For those institutions conducting computer science curricula based on current ACM standards, the software engineering curriculum track can be implemented easily. Implementation issues are much more manageable, including the important considerations that must be given to course scheduling, faculty preparation, student loads, hardware and software resources, instructional materials and curriculum development.

Section 3: The Software Engineering Transfer Curriculum Track

The *Computing Curricula 2003: Guidelines for Associate-Degree Curricula in Computer Science* report details a variety of paradigms for introductory computer science curricula, together with computer science and mathematics course descriptions. Two of these paradigms – Imperative-first and Objects-first – are suitable in a software engineering curriculum. The tables below outline a two-year software engineering curriculum track built upon each of those two computer science paradigms. The descriptions of the computer science and mathematics courses identified below are detailed in Appendices A, B, and C; the description for the SE201 software engineering course is detailed below.

The use of computing and mathematics courses, as well as overall curriculum structure, from the *Computing Curricula 2003: Guidelines for Associate-Degree Curricula in Computer Science* report enables two-year colleges to incorporate a software engineering curriculum track easily into an existing computer science transfer degree program. Students who complete this track could reasonably expect to transfer into baccalaureate software engineering programs consistent with the *Software Engineering 2004: Curriculum Guidelines for Undergraduate Degree Programs in Software Engineering*.

Year One

Imperative-First Paradigm

First Semester	Second Semester
CS101I (Programming Fundamentals)	**CS102I** (The Object-Oriented Paradigm**)**
	CS105 (Discrete Structures I)

Object-First Paradigm

First Semester	Second Semester
CS101O (Introduction to Object-Oriented Programming)	**CS102O** (Objects and Data ion)
	CS105 (Discrete Structures I)

Year Two

Imperative-First Paradigm

Third Semester	Fourth Semester
CS103I (Data Structures and Algorithms)	**SE201 (**Introduction to Software Engineering)
CS106 (Discrete Structures II)	

Object-First Paradigm

Third Semester	Fourth Semester
CS103O (Algorithms and Data Structures)	**SE201 (**Introduction to Software Engineering)
CS106 (Discrete Structures II)	

Table 1: Software Engineering Transfer Curriculum Track

Three things should be noted after review of Table 1 above:

- The software engineering track fits very well into the computer science transfer degree program, and the SE201 course can simply take the place of a suggested second-year elective course.
- There is no impact on or addition to a student's initial computer science sequence of study.
- Students interested in the field of software engineering can simply be added to the existing computer science and mathematics courses.

The following information details the SE201 course description, syllabus, student performance objectives, and sample laboratory experiences. As described herein, this course is consistent with the SE201 course described in the *Software Engineering 2004: Curriculum Guidelines for Undergraduate Degree Programs in Software Engineering* report. This will assist in the development of articulation agreements and student transfer between associate-degree granting institutions and baccalaureate-degree granting institutions.

<u>**SE201 Introduction to Software Engineering**</u>

This core course introduces the basic principles and concepts of software engineering and provides the necessary foundation for subsequent SE courses at the upper division level. Topics include: basic terminology and concepts of software engineering; system requirements, modeling, and testing; object oriented analysis and design using UML; frameworks and APIs; client-server architecture; user interface technology; and the analysis, design and programming of simple servers and clients.

Prerequisite: CS102I or CS102O

Student Performance Objectives:
Upon completion of this course, students will be able to:

- Develop clear, concise, and sufficiently formal requirements for extensions to an existing system, based on the true needs of users and other stakeholders.
- Identify software engineering tools, their uses, and benefits derived from the use of Computer Aided Systems Engineering (CASE).
- Apply design principles and patterns on reusable technology while designing and implementing simple distributed systems, and differentiate between structured design and object-oriented design.
- Create UML class diagrams which model aspects of the domain and the software architecture, and UML sequence diagrams and state machines that correctly model system behavior.
- Implement simple graphical user interfaces for a system, and apply simple measurement techniques to software.
- Demonstrate an appreciation for the breadth of software engineering, including the role of a software engineer and the associated ethical considerations.

Syllabus:

- Software engineering and its place as an engineering discipline.
- Review of the principles of object orientation.
- Reusable technologies as a basis for software engineering: frameworks and APIs.
- Introduction to client-server computing.
- Requirements analysis.
- UML class diagrams and object-oriented analysis; introduction to formal modeling using OCL.
- Examples of building class diagrams to model various domains.
- Design patterns (ion-occurrence, composite, player-role, singleton, observer, delegation, façade, adapter, observer, etc.).
- Use cases and user-centered design.
- Representing software behavior: sequence diagrams, state machines, activity diagrams.
- General software design principles: decomposition, decoupling, cohesion, reuse, reusability, portability, testability, flexibility, etc.
- Software architecture: distributed architectures, pipe-and-filter, model-view-controller, etc.
- Introduction to testing and project management.

Sample labs and assignments:

- Evaluating the performance of various simple software designs.
- Adding features to an existing system.
- Testing a system to verify conformance to test cases.
- Building a GUI for an application.
- Numerous exercises building models in UML, particularly class diagrams and state machines.
- Developing and presenting a simple set of requirements (to be done as a team) for some innovative client server application of very small size.
- Implementing the above, using reusable technology to the greatest extent possible.

Additional teaching considerations:

- This course is a good starting point for exposing students to moderately sized existing systems. With such systems, they can learn and practice the essential skills of reading and understanding code written by others. Students should write code in the context of a particular domain, for example the biological, physical, mathematical or chemical sciences or even wider spectra such as game programming, business applications, and graphics and animation.
- It is assumed that students entering this course will have had little coverage of software engineering concepts previously, but have had two courses that give them a very good background in programming and basic computer science. It is suggested that a core subset of UML be taught, rather than trying to cover all features.
- It may be challenging for instructors to convey the nature of SE to students; however, this challenge may be addressed through strategies such as field trips to businesses and industries that utilize large software systems, guest lectures by developers and users of large software systems, and discussions about embedded systems in everyday life including ATMs, wireless devices, cell phones, PDAs, portable MP3 players, and computer games.

Section 4: Additional Program Considerations

The mathematics of discrete structures underlies all computing fields, including software engineering. Hence, the mathematics courses CS105-106 (Discrete Structures I, II) identified in this report are core to the software engineering curriculum track. While these courses are sufficient to support the CS101-102-103-SE201 curriculum described in this report, they can be meaningfully supplemented by an additional course devoted to statistics and empirical methods. Not dissimilar from a statistics course offered frequently in the two-year college setting, such a course may be necessary for the upper division software engineering curriculum at some transfer institutions. The *Software Engineering 2004: Curriculum Guidelines for Undergraduate Degree Programs in Software Engineering* report identifies this potential need and addresses it through a course referred to as MA271 (Statistics and Empirical Methods). The course description reads "Applied probability and statistics in the context of computing; experiment design and the analysis of results; taught using examples from software engineering and other computing disciplines." It should also be noted that in order to fulfill articulation agreements with some transfer institutions students may also need to complete a calculus sequence (or additionally, linear algebra and/or differential equations).

Laboratory science courses such as physics, chemistry and biology provide students with content knowledge and experience with the scientific method (summarized as formulating problem statements and hypothesizing, designing and conducting experiments, observing and collecting data, analyzing and reasoning, and evaluating and concluding). Program requirements of this nature provide students with a foundation should they later develop software in those scientific domains. It should be noted that in some instances students may be required to complete a laboratory science course sequence as part of this degree program in order to gain entry into the upper division.

Some two-year colleges offer introductory engineering courses, providing an overview of the many individual disciplines constituting the world of engineering. These courses often engage students in stimulating activities that peak their interests and set the stage for career choices in such fields. Students pursuing software engineering degree programs would strengthen their insights into engineering by completing such coursework.

In their upper division work, students will focus their emerging software engineering skills in a particular application area of interest to them. The foundation for that selection may be laid in various elective courses that students pursue in the lower division. These could include courses in business and finance; biology and health sciences; mathematics and statistics; and information technology.

Effective oral and written communications abilities are of critical importance to software engineering professionals; therefore, students should be required to complete communications courses as part of this degree program. These skills must be identified, developed, nurtured and incorporated throughout a software engineering curriculum. Students must master effective writing, speaking, and listening abilities, and then consistently demonstrate those talents in a variety of settings, including formal and informal, large group and one-on-one, technical and non-technical, point and counter-point.

Colleges will ensure that degree programs ultimately fulfill all general education and related requirements arising from institutional, state, and regional accreditation guidelines. The curriculum recommendations contained herein are compatible with those requirements. Articulation agreements often guide curriculum content as well, and are important considerations in the formulation of programs of study, especially for transfer-oriented programs.

Professional software engineers have a responsibility to society and their work carries significant liabilities. Consequently, software engineers must conduct themselves in an ethical and professional manner. The preamble to the *Software Engineering Code of Ethics and Professional Practice* [ACM 1999] 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.*

Hence, instructors must ensure that the software engineering curriculum forces students to become familiar with the *Cod*e, and engages them in discussions and activities that emphasize the eight principles of the *Code*.

There is an alternate approach to the computing curriculum sequence outlined in this report that would place students into a software engineering course sequence at the onset, in advance of the computer science coursework. This approach is detailed in the *Software Engineering 2004: Curriculum Guidelines for Undergraduate Degree Programs in Software Engineering* report. With this approach, in the first year students take two courses (SE101, then SE102) that focus on software engineering with a major emphasis on the engineering perspective, but also introduce some programming and fundamental computer science concepts. In the second year, students take two courses (CS103 and SE200) that complete the development of the computer science content. For associate degree granting institutions with existing computer science programs, this alternative approach is not the most feasible implementation; for other institutions, the alternative approach may be feasible.

Bibliography

ACM Two-Year College Computing Curricula Task Force, *Guidelines for Associate-Degree Programs in Information Systems,* ACM Press (2004).

ACM Two-Year College Computing Curricula Task Force, *Computing Curricula 2003: Guidelines for Associate-Degree Curricula in Computer Science*, ACM Press (2003).

ACM Two-Year College Computing Curricula Task Force, *Computing Curricula Guidelines for Associate-Degree Programs: Computing Sciences*. ACM Press (1993).

Association for Computing Machinery, Inc. & the Institute for Electrical and Electronics Engineers, Inc. *Software Engineering Code of Ethics and Professional Practice.* (1999). Retrieved July 12, 2005 from http://www.acm.org/serving/se/code.htm.

Bloom, Benjamin S., *the Taxonomy of Educational Objectives: Classification of Educational Goals. Handbook I: The Cognitive Domain*, McKay Press, New York (1956).

Gorgone, Davis, Valacich, Topi, Feinstein, and Longnecker. *IS 2002 Model Curriculum and Guidelines for Undergraduate Degree Programs in Information Systems*. Association for Computing Machinery, et al. (2002).

IEEE-CS/ACM Joint Curriculum Task Force, *Software Engineering 2004: Curriculum Guidelines for Undergraduate Degree Programs in Software Engineering.*

IEEE-CS/ACM Joint Curriculum Task Force, *Computing Curricula 2001: Computer Science.*

Appendix A
Computer Science Imperative-First Course Descriptions

The Imperative-first approach consists of a three-course sequence that begins with a procedural structured-programming approach to fundamental programming concepts, followed by object-oriented concepts, and culminates with data structures.

At the completion of the CS101$_I$, CS102$_I$, CS103$_I$ course sequence, the following student performance objectives will be met. These performance objectives are grouped by the body of knowledge categories for computer science.

AL: Algorithms and Complexity Student Performance Objectives

Basic algorithmic analysis

1. Explain the use of big O, omega, and theta notation to describe the amount of work done by an algorithm.
2. Determine the time and space complexity of simple algorithms.

Algorithmic strategies

1. Describe the shortcoming of brute-force algorithms.
2. Implement a divide and conquer algorithm like Quicksort.
3. Discuss assorted heuristic problem solving methods.

Fundamental computing algorithms

1. Design and implement various quadratic and O(NlogN) sorting algorithms.
2. Design and implement an appropriate hashing function for an application.
3. Discuss the efficiency considerations for sorting searching and hashing.
4. Design and implement a collision-resolution algorithm for a hash table.
5. Discuss other performance considerations such as small versus large files, programming time, etc.

Basic computability

1. Provide a sample problem that has no algorithmic solution.

AR: Architecture and Organization Student Performance Objectives

Machine level representation of data

1. Explain the purpose of different formats to represent numerical data.
2. Explain how negative integers are stored in sign-magnitude and two’s-complement representation.
3. Describe the internal representation of non-numeric data.

Assembly level machine organization

1. Explain the organization of the classical von Neumann machine and its major functional units.
2. Explain how to execute an instruction in a classical von Neumann machine.

GV: Graphics and Visual Computing Student Performance Objectives

Fundamental techniques in graphics

1. Distinguish the capabilities of different levels of graphics software and describe the appropriateness of each.
2. Produce images using a standard graphics API.
3. Discuss the 3-dimensional world coordinate system.

HC: Human Computer Interaction Student Performance Objectives

Foundations of HCI

1. Discuss the reasons for human-centered software development.
2. Summarize the basic science of psychological and social interaction.
3. Distinguish between the different interpretations that the same icon, symbol, word, and color have among varying human cultures.
4. Identify ways to respect human diversity when interacting with a computer system.

Building simple GUI

1. Identify several fundamental principles for effective GUI design.
2. Use a GUI toolkit to create a simple application that supports a graphical user interface.
3. Produce two instances of the same GUI design; one based on fundamental design principles and the other ignoring these principles.
4. Conduct a simple usability test for each instance and compare the results.

IM: Information Management Student Performance Objectives

Database systems

1. Explain the characteristics that distinguish the database approach from the traditional approach of programming with data files.
2. Describe the components of a database system and give examples of their use.

NC: Net-Centric Computing Student Performance Objectives

Introduction to net-centric computing

1. Discuss the evolution of early networks and the Internet.
2. Describe emerging technologies in the net-centric computing area such as wireless computing and voice over IP.

OS: Operating Systems Student Performance Objectives

Overview of operating systems

1. Explain the objectives and functions of modern operating systems.
2. Describe how operating systems historically have evolved from primitive batch systems to sophisticated multi-user systems.
3. Describe the functions of a contemporary operating system with respect to convenience, efficiency, and the ability to evolve.

PF: Programming Fundamentals Student Performance Objectives

Fundamental programming constructs

1. Analyze and explain the behavior of simple programs involving the fundamental programming constructs.
2. Explain the use of each data type and how each is stored in memory.
3. Modify and expand short programs using control structures and functions.
4. Design, implement, test and debug a program that uses each of the following fundamental programming constructs: basic computation, simple I/O, standard conditional and iterative structures, and the definition of functions.
5. Choose appropriate selection and iteration constructs for a given programming task.
6. Apply the techniques of structured (functional) decomposition to break a program into smaller pieces.
7. Describe parameters passing between functions.

Algorithms and problem solving

1. Discuss why algorithms are useful in problem solving with a programming language.
2. List the recommended steps in problem solving.
3. Create algorithms for solving simple problems.
4. Use pseudocode or a programming language to implement, test, and debug algorithms for problem solving.
5. Discuss what makes a good algorithm.
6. Analyze an algorithm's correctness and efficiency.

Fundamental data structures

1. Define a data structure and an Data Type (ADT) and distinguish among built-in and user-defined data structures.