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

August 2005

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. 0003263.

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

www.acmtyc.org

hold this page for copyright information

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 http://www.computer.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 http://www.computer.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 abstraction 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 Abstraction)
	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.

SE201 Introduction to Software Engineering

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 (abstraction-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 Abstract Data Type (ADT) and distinguish among built-in and user-defined data structures.

2. Discuss the representation and use of primitive data types and built-in data structures.
3. Describe common applications for each data structure covered.
4. Describe ADTs at a logical or abstract level and discuss how each works.
5. Implement the user-defined data structures in a high-level language.
6. Compare alternative implementations of data structures with respect to performance.
7. Write and execute a program for testing a data structure implementation.
8. Compare and contrast dynamic and static data structure implementations.
9. Choose the appropriate data structure for modeling a given problem.

Recursion

1. Describe and exemplify the concept of recursion.
2. Verify correctness of a recursive routine by identifying the base case and the general case.
3. Compare iterative and recursive solutions for elementary problems such as factorial.
4. Define the divide-and-conquer approach.
5. Implement, test, and debug simple recursive functions.
6. Describe how recursion can be implemented using a stack.

Event-driven programming

1. Explain the difference between event-driven programming and command-line programming.
2. Identify programming languages that support event-driven programming and those that do not.
3. Produce code to test and debug simple event-driven programs that respond to user events.

PL: Programming Languages Student Performance Objectives

Overview of programming languages

1. Summarize the evolution of programming languages illustrating how this history has led to the paradigms available today.
2. Identify at least one distinguishing characteristic for each of the programming paradigms covered in this unit.
3. Evaluate the tradeoffs between the different paradigms, considering such issues as space efficiency, time efficiency (of both the computer and the programmer), safety, and power of expression.
4. Define issues concerning programming-in-the-small versus programming-in-the-large.

Virtual machines

1. Describe the importance and power of abstraction in the context of virtual machines.
2. Explain the benefits of intermediate languages in the compilation process.
3. Evaluate the tradeoffs in performance vs. portability.
4. Explain how executable programs can breach computer system security by accessing disk files and memory.

Introduction to language translation

1. Compare and contrast compiled and interpreted execution models, outlining the relative merits of each.
2. Describe the phases of program translation from source code to executable code and the files produced by these phases.

Declaration and types

1. Explain the value of declaration models, especially with respect to programming-in-the-large.
2. Identify the properties of a variable [object] such as its associated address, value, scope, persistence, and size.
3. Discuss type incompatibility.
4. Demonstrate different forms of binding, visibility, scoping, and lifetime management.
5. Defend the importance of types and type checking in providing abstraction and safety.
6. Evaluate tradeoffs in lifetime management (reference counting vs. garbage collection).

Abstraction mechanisms

1. Explain how abstraction mechanisms support the creation of reusable software components.
2. Demonstrate the difference between call-by-value and call-by-reference parameter passing.
3. Defend the importance of abstractions, especially with respect to programming-in-the-large.
4. Describe how the computer system uses activation records to manage program modules and their data.

Object-oriented programming

1. Justify the philosophy of object-oriented design and the concepts of encapsulation, abstraction, inheritance, and polymorphism.
2. Design, implement, test, and debug simple programs in an object-oriented programming language.
3. Defend the benefits of separating class specification from class implementation

4. Describe how the class mechanism supports encapsulation and information hiding.
5. Design, implement, and test the implementation of "is-a" relationships among objects using a class hierarchy and inheritance.
6. Implement polymorphism using virtual functions.
7. Compare and contrast the notion of [static] overloading v. [run-time] overriding functions.
8. Describe the [different levels] use of selective inheritance.
9. Demonstrate the use of generic program components that use types as parameters.
10. Explain the relationship between the static structure of the class and the dynamic structure of the instances [objects] of the class.
11. Describe how iterators access the elements of a container.

SE: Software Engineering Student Performance Objectives

Software design

1. Discuss the properties of good software design.
2. Compare and contrast object-oriented analysis and design with structured analysis and design.
3. Select and apply appropriate design patterns in the construction of a software application.
4. Create and specify the software design for a medium-size software product using a software requirement specification and the Unified Modeling Language (UML).

Using APIs

1. Explain the value of application programming interfaces (APIs) in software development.
2. Explain the difference between sequential and event-driven programming.
3. Use class browsers and related tools during the development of applications using APIs.
4. Examine programs that use large-scale API packages.

Software tools and environments

1. Select, with justification, an appropriate set of CASE tools to support the software development of a range of software products.
2. Analyze and evaluate a set of tools in a given area of software development (e.g., management, modeling, or testing).
3. Demonstrate the use of a range of software tools in support of the development of a software product of medium size.

Software processes

1. Explain the software life cycle, define its phases, and describe the deliverables that are produced.
2. Explain the role of process maturity models.
3. Describe the software engineering process using standard metrics.
4. Compare the traditional waterfall model to the incremental model, the object-oriented model, and other select models.

Software requirements and specifications

1. Apply key elements and common methods for elicitation and analysis to produce a set of software requirements for a medium-sized software system.
2. Use a common, non-formal method to model and specify (in the form of a requirements specification document) the requirements for a medium-size software system (e.g., structured analysis or object-oriented-analysis).
3. Review a software requirements document using best practices to determine its quality.
4. Demonstrate a commonly used prototyping tool.

Software validation

1. Distinguish between program validation and verification.
2. Distinguish between and implement the different types and levels of testing (unit, integration, systems, and acceptance) for medium-size software products.
3. Analyze a test plan for a medium-size code segment.
4. Evaluate, as part of a team, an inspection of a medium-size code segment.
5. Describe the role that tools can play in the validation of software.

Software evolution

1. Identify the principal issues associated with software evolution and explain their impact on the software life cycle.
2. Discuss the advantages and disadvantages of software reuse.
3. Discuss the opportunities for software reuse in a variety of contexts.

Software project management

1. Summarize the key elements of team building and team management.
2. Review a software project plan that includes topics such as: estimates of size and effort, a schedule, resource allocation, configuration control, change management, and project risk identification and management.
3. Compare and contrast the different methods and techniques used to assure the quality of a software product.

SP: Social and Professional Student Performance Objectives

History of computing

1. List the contributions of several pioneers in the computing field.
2. Compare daily life before and after the advent of computing.
3. Identify significant continuing trends in the history of the computing field and their impact on society.

Social context of computing

1. Interpret the social context of a particular computing implementation.
2. Identify assumptions and values embedded in a particular design.
3. Describe ways in which computing alters the interaction between people.
4. Explain why computing/network access is restricted in certain countries.

Professional and ethical responsibility

1. Name the strengths and weaknesses of relevant professional codes as expressions of professionalism and guides to decision-making.
2. Identify ethical issues that arise in software development and determine how to address them technically and ethically.
3. Review a computer use policy with enforcement measures.
4. Discuss the professional codes of ethics from the ACM, the IEEE Computer Society, and other organizations.

Intellectual property

1. Distinguish among software patent, copyright, and trade secret protection including statue of limitation.
2. Discuss the legal background of copyright in national and international law.
3. Recognize that patent and copyright laws may vary from country to country.
4. Discuss the consequences of software piracy on software developers and the role of enforcement organizations.

CS101I: Programming Fundamentals

This course 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.

Prerequisites: No programming or computer science experience is required; mathematical preparation sufficient to qualify for precalculus at the college level.

Syllabus:

- Computing applications: Word processing; spreadsheets; editors; files and directories
- Fundamental programming constructs: Syntax and semantics of a higher-level language; variables, types, expressions, and assignment; simple I/O; conditional and iterative control structures; functions and parameter passing; structured decomposition
- Algorithms and problem-solving: Problem-solving strategies; the role of algorithms in the problem-solving process; implementation strategies for algorithms; debugging strategies; the concept and properties of algorithms
- Fundamental data structures: Primitive types; arrays; records; strings and string processing
- Machine level representation of data: Bits, bytes, and words; numeric data representation and number bases; representation of character data
- Overview of operating systems: The role and purpose of operating systems; simple file management
- Introduction to net-centric computing: Background and history of networking and the Internet; demonstration and use of networking software including e-mail, telnet, and FTP
- Human-computer interaction: Introduction to design issues
- Software development methodology: Fundamental design concepts and principles; structured design; testing and debugging strategies; test-case design; programming environments; testing and debugging tools
- Social context of computing: History of computing and computers; evolution of ideas and machines; social impact of computers and the Internet; professionalism, codes of ethics, and responsible conduct; copyrights, intellectual property, and software piracy.

Knowledge Area	***Units Covered***	***Suggested Hours***
PF	Fundamental programming constructs	10
	Algorithms and problem-solving	3
	Fundamental data structures	2
PL	Introduction to language translation	1
	Declarations and types	3
	Abstraction mechanisms	3
AR	Machine level representation of data	1
	Assembly level machine organization	2
OS	Overview of operating systems	1
NC	Introduction to net-centric computing	1
HC	Foundations of human-computer interaction	1
GV	Fundamental techniques in graphics	1
SE	Software design	3
	Software tools and environments	2
	Software processes	1
SP	History of computing	1
	Social context of computing	1
	Professional and ethical responsibilities	1
	Intellectual property	1
Other	Topics of local interest	1
	TOTAL:	**40**

Table 2: CS101$_I$, Programming Fundamentals

Notes:

This course represents the first semester of an imperative-first introductory track that covers fundamental programming concepts in three semesters. Although covering programming fundamentals in two semesters has long been standard in computer science education, more and more programming topics can legitimately be identified as fundamental, making it more difficult to provide a complete introduction to this material in a single year. In terms of the curriculum, students should be able to move on to more advanced computer science courses after taking course sequence CS101I-102I-103I.

Although this is the first course of the educational process for a computer science student, it is reasonable to expect that a student will have at least some exposure to computers before taking this course. The prepared student will have experience with email, World Wide Web use, and basic word processing. Note that this course will not necessarily be taken during a student's first semester in college. Any remedial work (generally identified as "developmental studies" or "learning support" coursework) in mathematics or language arts should be completed before the student is allowed to begin this course sequence.

The intent of this introductory course will be to contain an integrated laboratory component. The lab component provides the important hands-on programming experience that is vital for beginning computer science students. The forty (40) total hours listed above are considered "lecture" hours and typically two (2) hours of lab equates to one (1) hour of lecture.

CS102I: The Object-Oriented Paradigm

This course 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.

Prerequisite: CS 101I
Corequisite: CS 105

Syllabus:

- Review of control structures, functions, and primitive data types
- Object-oriented programming: Object-oriented design; encapsulation and information-hiding; separation of behavior and implementation; classes, subclasses, and inheritance; polymorphism; class hierarchies
- Fundamental computing algorithms: simple searching and sorting algorithms (linear and binary search, selection and insertion sort)
- Fundamentals of event-driven programming
- Introduction to computer graphics: Using a simple graphics API
- Overview of programming languages: History of programming languages; brief survey of programming paradigms
- Virtual machines: The concept of a virtual machine; hierarchy of virtual machines; intermediate languages
- Introduction to language translation: Comparison of interpreters and compilers; language translation phases; machine-dependent and machine-independent aspects of translation
- Introduction to database systems: History and motivation for database systems; use of a database query language
- Software evolution: Software maintenance; characteristics of maintainable software; reengineering; legacy systems; software reuse

Knowledge Area	***Units Covered***	***Suggested Hours***
PF	Fundamental programming constructs	3
	Algorithms and problem-solving	6
	Fundamental data structures	5
	Event-driven programming	1
AL	Fundamental computing algorithms	3
PL	Overview of programming languages	1
	Virtual machines	1
	Introduction to language translation	1
	Object-oriented programming	6
AR	Machine level representation of data	2
HC	Foundations of human computer interaction	1
	Building a simple graphical user interface	2
IM	Database systems	1
SE	Software design	1
	Using APIs	2
	Software requirements and specifications	1
	Software validation	1
	Software evolution	1
Other	Topics of local interest	1
	TOTAL:	**40**

Table 3: CS102$_{I}$, the Object-Oriented Paradigm

Notes:

This course represents the second semester of an imperative-first introductory track that covers fundamental programming concepts in three semesters. Although covering programming fundamentals in two semesters has long been standard in computer science education, more and more programming topics can legitimately be identified as fundamental, making it more difficult to provide a complete introduction to this material in a single year. In terms of the curriculum, students should be able to move on to more advanced computer science courses after taking course sequence CS101I-102I-103I.

This second computer science course should be taken in parallel with the first discrete mathematics course, CS 105, to ensure that the appropriate mathematical foundations of computer science are covered along with the applications of those topics.

The intent of this introductory course will be to contain an integrated laboratory component. The lab component provides the important hands-on programming experience that is vital for beginning computer science students. The forty (40) total hours listed above are considered "lecture" hours and typically two (2) hours of lab equates to one (1) hour of lecture.

CS103I: Data Structures and Algorithms

This course builds upon the foundation provided by the CS101I-102I 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.

Prerequisites: CS 102I
Corequisite: CS 106

Syllabus:

- Review of elementary programming concepts
- Fundamental data structures: Stacks; queues; linked lists; hash tables; trees; graphs
- Object-oriented programming: Object-oriented design; encapsulation and information hiding; classes; separation of behavior and implementation; class hierarchies; inheritance; polymorphism
- Fundamental computing algorithms: O(N log N) sorting algorithms; hash tables, including collision-avoidance strategies; binary search trees; representations of graphs; depth- and breadth-first traversals
- Recursion: The concept of recursion; recursive mathematical functions; simple recursive procedures; divide-and-conquer strategies; recursive backtracking; implementation of recursion
- Basic algorithmic analysis: Asymptotic analysis of upper and average complexity bounds; identifying differences among best, average, and worst case behaviors; big O, little o, omega, and theta notation; standard complexity classes; empirical measurements of performance; time and space tradeoffs in algorithms; using recurrence relations to analyze recursive algorithms
- Algorithmic strategies: Brute-force algorithms; greedy algorithms; divide-and-conquer; backtracking; branch-and-bound; heuristics; pattern matching and string/text algorithms; numerical approximation algorithms
- Overview of programming languages: Programming paradigms
- Software engineering: Software validation; testing fundamentals, including test plan creation and test case generation; object-oriented testing

Knowledge Area	***Units Covered***	***Suggested Hours***
PF	Fundamental data structures	12
	Recursion	5
AL	Basic algorithmic analysis	2
	Algorithmic strategies	3
	Fundamental computing algorithms	5
	Basic computability	1
PL	Overview of programming languages	1
	Object-oriented programming	8
SE	Software validation	1
	Software project management	1
Other	Topics of local interest	1
	TOTAL:	**40**

Table 4: CS103$_I$, Data Structures and Algorithms

Notes:

This course represents the third and final semester of an imperative-first introductory track that covers fundamental programming concepts in three semesters. Although covering programming fundamentals in two semesters has long been standard in computer science education, more and more programming topics can legitimately be identified as fundamental, making it more difficult to provide a complete introduction to this material in a single year. In terms of the curriculum, students should be able to move on to more advanced computer science courses after taking course sequence CS101I-102I-103I.

This third computer science course should be taken in parallel with the second discrete mathematics course, CS 106, to ensure that the appropriate mathematical foundations of computer science are covered along with the applications of those topics.

Appendix B
Computer Science Objects-First
Course Descriptions

The Objects-first implementation strategy incorporates throughout the curriculum object-oriented software design and programming methodologies. Object-oriented design promotes thinking about software development in a way that more closely models interaction with the real world. Modeling programming problems as abstract objects that communicate with each other helps to manage the complexity of software projects. The object-oriented programming paradigm is based on the relationship of interacting and cooperating data objects to solve computing problems.

At the completion of the $CS101_O$, $CS102_O$, $CS103_O$ 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. Use big O, omega, and theta notation to give asymptotic upper, lower, and tight bounds on time and space complexity of algorithms.
3. Determine the time and space complexity of simple algorithms.

Algorithmic strategies

1. Describe the shortcoming of brute-force algorithms.
2. Implement a greedy algorithm, such as Prim's algorithm.
3. Implement a divide and conquer algorithm like Quicksort.
4. Use backtracking to solve problem like mazes.
5. Discuss assorted heuristic problem solving methods.
6. Use pattern matching to analyze substrings.
7. Use numerical approximation to solve mathematical problems, such as finding the roots of a polynomial.

Fundamental computing algorithms

6. Design and implement various quadratic and *O*(NlogN) sorting algorithms.
7. Design and implement an appropriate hashing function for an application.
8. Discuss the efficiency considerations for sorting searching and hashing.
9. Design and implement a collision-resolution algorithm for a hash table.
10. 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. Discuss how fixed-length number representations affect accuracy and precision.
4. Describe the internal representation of non-numeric data.

GV: Graphics and Visual Computing 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.

Graphic systems

1. Describe the appropriateness of graphics architecture for a given application.
2. Explain the functions of different input devices.

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.

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. 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. Analyze an algorithm's correctness and efficiency.

Fundamental data structures

1. Define a data structure and an Abstract Data Type (ADT) and distinguish among built-in and user-defined data structures.
2. Discuss the representation and use of primitive data types and built-in data structures.
3. Describe common applications for each data structure covered.
4. Describe ADTs at a logical or abstract level and discuss how each works.
5. Implement the user-defined data structures in a high-level language.
6. Compare alternative implementations of data structures with respect to performance.
7. Write and execute a program for testing a data structure implementation.
8. Compare and contrast dynamic and static data structure implementations.
9. Choose the appropriate data structure for modeling a given problem.

Recursion

1. Describe and exemplify the concept of recursion.
2. Verify correctness of a recursive routine by identifying the base case and the general case.
3. Compare iterative and recursive solutions for elementary problems such as factorial.
4. Define the divide-and-conquer approach.

5. Implement, test, and debug simple recursive functions.
6. Describe how recursion can be implemented using a stack.
7. Discuss problems for which backtracking is an appropriate solution.
8. Determine when a recursive solution is appropriate for a problem.

Event-driven programming

1. Explain the difference between event-driven programming and command-line programming.
2. Identify programming languages that support event-driven programming and those that do not.
3. Produce code to test and debug simple event-driven programs that respond to user events.
4. Produce code that responds to exception conditions raised during execution.

PL: Programming Languages Student Performance Objectives

Overview of programming languages

1. Summarize the evolution of programming languages illustrating how this history has led to the paradigms available today.
2. Identify at least one distinguishing characteristic for each of the programming paradigms covered in this unit.
3. Evaluate the tradeoffs between the different paradigms, considering such issues as space efficiency, time efficiency (of both the computer and the programmer), safety, and power of expression.
4. Define issues concerning programming-in-the-small versus programming-in-the-large.

Virtual machines

1. Describe the importance and power of abstraction in the context of virtual machines.
2. Explain the benefits of intermediate languages in the compilation process.
3. Evaluate the tradeoffs in performance vs. portability.
4. Explain how executable programs can breach computer system security by accessing disk files and memory.

Declaration and types

1. Explain the value of declaration models, especially with respect to programming-in-the-large.
2. Identify the properties of a variable [object] such as its associated address, value, scope, persistence, and size.
3. Discuss type incompatibility.
4. Demonstrate different forms of binding, visibility, scoping, and lifetime management.

5. Defend the importance of types and type checking in providing abstraction and safety.
6. Evaluate tradeoffs in lifetime management (reference counting vs. garbage collection).

Abstraction mechanisms

1. Explain how abstraction mechanisms support the creation of reusable software components.
2. Demonstrate the difference between call-by-value and call-by-reference parameter passing.
3. Defend the importance of abstractions, especially with respect to programming-in-the-large.
4. Describe how the computer system uses activation records to manage program modules and their data.

Object-oriented programming

1. Justify the philosophy of object-oriented design and the concepts of encapsulation, abstraction, inheritance, and polymorphism.
2. Design, implement, test, and debug simple programs in an object-oriented programming language.
3. Defend the benefits of separating class specification from class implementation
4. Describe how the class mechanism supports encapsulation and information hiding.
5. Design, implement, and test the implementation of “is-a” relationships among objects using a class hierarchy and inheritance.
6. Implement polymorphism using virtual functions.
7. Compare and contrast the notion of [static] overloading v. [run-time] overriding functions.
8. Describe the [different levels] use of selective inheritance.
9. Demonstrate the use of generic program components that use types as parameters.
10. Explain the relationship between the static structure of the class and the dynamic structure of the instances [objects] of the class.
11. Describe how iterators access the elements of a container.

Functional Programming

1. Outline the strengths and weaknesses of the functional programming paradigm.
2. Analyze programs that use the functional paradigm.

SE: Software Engineering Student Performance Objectives

Software design

1. Discuss the properties of good software design.
2. Compare and contrast object-oriented analysis and design with structured analysis and design.
3. Select and apply appropriate design patterns in the construction of a software application.
4. Create and specify the software design for a medium-size software product using a software requirement specification and the Unified Modeling Language (UML).

Using APIs

1. Explain the value of application programming interfaces (APIs) in software development.
2. Explain the difference between sequential and event-driven programming.
3. Use class browsers and related tools during the development of applications using APIs.
4. Examine programs that use large-scale API packages.

Software tools and environments

1. Select, with justification, an appropriate set of CASE tools to support the software development of a range of software products.
2. Analyze and evaluate a set of tools in a given area of software development (e.g., management, modeling, or testing).
3. Demonstrate the use of a range of software tools in support of the development of a software product of medium size.

Software processes

1. Explain the software life cycle, define its phases, and describe the deliverables that are produced.
2. Explain the role of process maturity models.
3. Describe the software engineering process using standard metrics.
4. Compare the traditional waterfall model to the incremental model, the object-oriented model, and other select models.

Software requirements and specifications

1. Apply key elements and common methods for elicitation and analysis to produce a set of software requirements for a medium-sized software system.
2. Use a common, non-formal method to model and specify (in the form of a requirements specification document) the requirements for a medium-size software system (e.g., structured analysis or object-oriented-analysis).
3. Review a software requirements document using best practices to determine its quality.
4. Demonstrate a commonly used prototyping tool.

Software validation

1. Distinguish between program validation and verification.
2. Distinguish between and implement the different types and levels of testing (unit, integration, systems, and acceptance) for medium-size software products.
3. Analyze a test plan for a medium-size code segment.
4. Evaluate, as part of a team, an inspection of a medium-size code segment.
5. Describe the role that tools can play in the validation of software.

Software evolution

1. Identify the principal issues associated with software evolution and explain their impact on the software life cycle.
2. Discuss the advantages and disadvantages of software reuse.
3. Discuss the opportunities for software reuse in a variety of contexts.

Software project management

1. Summarize the key elements of team building and team management.
2. Review a software project plan that includes topics such as: estimates of size and effort, a schedule, resource allocation, configuration control, change management, and project risk identification and management.
3. Compare and contrast the different methods and techniques used to assure the quality of a software product.

SP: Social and Professional Student Performance Objectives

History of computing

1. List the contributions of several pioneers in the computing field.
2. Compare daily life before and after the advent of computing.
3. Identify significant continuing trends in the history of the computing field and their impact on society.

Social context of computing

1. Interpret the social context of a particular computing implementation.
2. Identify assumptions and values embedded in a particular design.
3. Describe ways in which computing alters the interaction between people.
4. Explain why computing/network access is restricted in certain countries.

Professional and ethical responsibility

1. Name the strengths and weaknesses of relevant professional codes as expressions of professionalism and guides to decision-making.
2. Identify ethical issues that arise in software development and determine how to address them technically and ethically.
3. Review a computer use policy with enforcement measures.

4. Discuss the professional codes of ethics from the ACM, the IEEE Computer Society, and other organizations.

Risks and liabilities of computer-based systems

1. Explain, using case studies, the limitations of software testing.
2. Describe the differences between correctness, reliability, and safety.
3. Discuss the potential for hidden problems in reuse of existing components.
4. Recognize the risks associated with various computer crimes.

CS101O: Introduction to Object-Oriented Programming

This course introduces the fundamental concepts of programming from an object-oriented perspective. Topics include simple data types, control structures, an introduction to array and string data structures and algorithms, as well as debugging techniques and the social implications of computing. The course emphasizes good software engineering principles and developing fundamental programming skills in the context of a language that supports the object-oriented paradigm.

Prerequisites: No programming or computer science experience is required; mathematical preparation sufficient to qualify for precalculus at the college level.

Syllabus:

- Introduction to the history of computer science
- Ethics and responsibility of computer professionals
- Introduction to computer systems and environments
- Introduction to object-oriented paradigm: Abstraction; objects; classes; methods; parameter passing; encapsulation; inheritance; polymorphism
- Fundamental programming constructs: Basic syntax and semantics of a higher-level language; variables, types, expressions, and assignment; simple I/O; conditional and iterative control structures; structured decomposition
- Fundamental data structures: Primitive types; arrays; records; strings and string processing
- Introduction to programming languages
- Algorithms and problem-solving: Problem-solving strategies; the role of algorithms in the problem-solving process; implementation strategies for algorithms; debugging strategies; the concept and properties of algorithms

Knowledge Area	***Units Covered***	***Suggested Hours***
PF	Fundamental programming constructs	9
	Algorithms and problem-solving	4
	Fundamental data structures	3
AL	Fundamental computing algorithms	1
	Basic computability	1
PL	Overview of programming languages	2
	Declarations and types	2
	Object-oriented programming	7
	Functional programming	1
AR	Machine level representation of data	2
SE	Software design	1
	Software tools and environments	1
	Software validation	1
SP	History of computing	1
	Social context of computing	1
	Professional and ethical responsibilities	1
	Risks and liabilities of computer-based systems	1
Other	Topics of local interest	1
	TOTAL:	**40**

Table 5: CS101$_O$, Introduction to Object-Oriented Programming

Notes:

This course represents the first semester of an objects-first introductory track that covers fundamental programming concepts in three semesters. Although covering programming fundamentals in two semesters has long been standard in computer science education, more and more programming topics can legitimately be identified as fundamental, making it more difficult to provide a complete introduction to this material in a single year. In terms of the curriculum, students should be able to move on to more advanced computer science courses after taking course sequence CS101O-102O-103O.

Although this is the first course of the educational process for a computer science student, it is reasonable to expect that a student will have at least some exposure to computers before taking this course. The prepared student will have experience with email, World Wide Web use, and basic word processing. Note that this course will not necessarily be taken during a student's first semester in college. Any remedial work (generally identified as "developmental studies" or "learning support" coursework) in mathematics or language arts should be completed before the student is allowed to begin this course sequence.

What differentiates this course sequence from the imperative-first implementation in CS101I-102I 103I is the early emphasis on objects. The discussion of classes, subclasses,

and inheritance typically precedes even such basic concepts as conditional and iterative control statements.

The intent of this introductory course will be to contain an integrated laboratory component. The lab component provides the important hands-on programming experience that is vital for beginning computer science students. The forty (40) total hours listed above are considered “lecture” hours and typically two (2) hours of lab equates to one (1) hour of lecture.

CS102O: Objects and Data Abstraction

This course continues the introduction from CS101O to the methodology of programming from an object-oriented perspective. Through the study of object design, this course also introduces the basics of human-computer interfaces, graphics, and the social implications of computing, with an emphasis on software engineering.

Prerequisite: CS 101O
Corequisite: CS 105

Syllabus:

- Review of object-oriented programming: Object-oriented methodology, object-oriented design; software tools
- Principles of object-oriented programming: Inheritance; class hierarchies; polymorphism; abstract and interface classes; container/collection classes and iterators
- Object-oriented design: Concept of design patterns and the use of APIs; modeling tools such as class diagrams, CRC cards, and UML use cases
- Virtual machines: The concept of a virtual machine; hierarchy of virtual machines; intermediate languages
- Fundamental computing algorithms: Searching; sorting; introduction to recursive algorithms
- Fundamental data structures: Built-in, programmer-created, and dynamic data structures
- Event-driven programming: Event-handling methods; event propagation; exception handling
- Foundations of human-computer interaction: Human-centered development and evaluation; principles of good design and good designers; engineering tradeoffs; introduction to usability testing
- Fundamental techniques in graphics: Hierarchy of graphics software; using a graphics API; simple color models; homogeneous coordinates; affine transformations; viewing transformation; clipping
- Software engineering issues: Tools; processes; requirements; design and testing; design for reuse; risks and liabilities of computer-based systems

Knowledge Area	***Units Covered***	***Suggested Hours***
PF	Fundamental data structures	3
	Recursion	2
	Event-driven programming	3
AL	Basic algorithmic analysis	1
	Fundamental computing algorithms	2
PL	Virtual machines	1
	Declarations and types	1
	Abstraction mechanisms	3
	Object-oriented programming	7
AR	Machine level representation of data	1
HC	Foundations of human-computer interaction	1
	Building a simple graphical user interface	1
GV	Fundamental techniques in graphics	2
	Graphic systems	1
SE	Software design	3
	Using APIs	2
	Software tools and environments	1
	Software requirements and specifications	1
	Software validation	1
	Software evolution	1
Other	Topics of local interest	2
	TOTAL:	**40**

Table 6: CS102$_O$, Objects and Data Abstraction

Notes:

This course represents the second semester of an objects-first introductory track that covers fundamental programming concepts in three semesters. Although covering programming fundamentals in two semesters has long been standard in computer science education, more and more programming topics can legitimately be identified as fundamental, making it more difficult to provide a complete introduction to this material in a single year. In terms of the curriculum, students should be able to move on to more advanced computer science courses after taking course sequence CS101O-102O-103O.

This second computer science course should be taken in parallel with the first discrete mathematics course, CS 105, to ensure that the appropriate mathematical foundations of computer science are covered along with the applications of those topics.

What differentiates this course sequence from the imperative-first implementation in CS101I-102I 103I is the early emphasis on objects. The discussion of classes, subclasses, and inheritance typically precedes even such basic concepts as conditional and iterative control statements.

The intent of this introductory course will be to contain an integrated laboratory component. The lab component provides the important hands-on programming experience that is vital for beginning computer science students. The forty (40) total hours listed above are considered "lecture" hours and typically two (2) hours of lab equates to one (1) hour of lecture.

CS103O: Algorithms and Data Structures

This course builds upon the introduction to object-oriented programming begun in CS101O and CS102O with an emphasis on algorithms, data structures, and software engineering.

Prerequisite: CS 102O
Corequisite: CS 106

Syllabus:

- Review of object-oriented design
- Review of basic algorithm design
- Review of professional and ethical issues
- Algorithms and problem solving: Classic techniques for algorithm design; problem solving in the object-oriented paradigm; application of algorithm design techniques to a medium-sized project, with an emphasis on formal methods of testing
- Basic algorithmic analysis: Asymptotic analysis of upper and average complexity bounds; identifying differences among best, average, and worst case behaviors; big O notation; standard complexity classes; empirical measurements of performance; time and space tradeoffs in algorithms
- Recursion: The concept of recursion; recursive mathematical functions; simple recursive procedures; divide-and-conquer strategies; recursive backtracking; implementation of recursion; recursion on trees and graphs
- Fundamental computing algorithms: Hash tables; binary search trees; representations of graphs; depth- and breadth-first traversals; shortest-path algorithms; transitive closure; minimum spanning tree; topological sort
- Fundamental data structures: Pointers and references; linked structures; implementation strategies for stacks, queues, and hash tables; implementation strategies for graphs and trees; strategies for choosing the right data structure
- Software engineering: Software project management; building a medium-sized system, in teams, with algorithmic efficiency in mind

Knowledge Area	***Units Covered***	***Suggested Hours***
PF	Algorithms and problem-solving	3
	Fundamental data structures	11
	Recursion	6
AL	Basic algorithmic analysis	3
	Algorithmic strategies	6
	Fundamental computing algorithms	5
SE	Software design	3
	Software project management	1
Other	Topics of local interest	2
	TOTAL:	**40**

Table 7: CS103$_O$, Algorithms and Data Structures

Notes:

This course represents the third and final semester of an objects-first introductory track that covers fundamental programming concepts in three semesters. Although covering programming fundamentals in two semesters has long been standard in computer science education, more and more programming topics can legitimately be identified as fundamental, making it more difficult to provide a complete introduction to this material in a single year. In terms of the curriculum, students should be able to move on to more advanced computer science courses after taking course sequence CS101O-102O-103O.

This third computer science course should be taken in parallel with the second discrete mathematics course, CS 106, to ensure that the appropriate mathematical foundations of computer science are covered along with the applications of those topics.

What differentiates this course sequence from the imperative-first implementation in CS101I-102I 103I is the early emphasis on objects. The discussion of classes, subclasses, and inheritance typically precedes even such basic concepts as conditional and iterative control statements.

The intent of this introductory course will be to contain an integrated laboratory component. The lab component provides the important hands-on programming experience that is vital for beginning computer science students. The forty (40) total hours listed above are considered “lecture” hours and typically two (2) hours of lab equates to one (1) hour of lecture.

Appendix C
Discrete Mathematics Course Descriptions

Students should complete a two-course discrete math sequence, as outlined below. The learning objectives associated with discrete mathematics support this degree program. A mathematics department or a computing department (or jointly) should deliver the courses with that intent.

At the completion of the CS105 and CS106 discrete math 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: Algorithmic and Complexity Student Performance Objectives

Basic algorithmic analysis

1. Use big O, omega, and theta notation to give asymptotic upper, lower, and tight bounds on time and space complexity of algorithms.
2. Deduce recurrence relations that describe the time complexity of recursively defined algorithms.

Basic computability

1. Discuss the concept of finite state machines.
2. Explain context-free grammars.
3. Provide examples that illustrate the implications of uncomputability.

Complexity of classes P and NP

1. Define the classes P and NP.
2. Explain the significance of NP-completeness.
3. Discuss a classic known NP-complete problem.

AR: Architecture and Organization

Digital logic and digital systems

1. Use mathematical expressions to describe the functions of simple combinational and sequential circuits.
2. Design a simple circuit using the fundamental building blocks.

DS: Discrete Structures Student Performance Objectives

Functions, relations, and sets

1. Explain with examples the basic terminology of functions, relations, and sets.
2. Perform the operations associated with sets, functions, and relations.
3. Relate practical examples to the appropriate set, function, or relation model, and interpret the associated operations and terminology in context.

4. Demonstrate basic counting principles, including uses of diagonalization and the pigeonhole principle.

Basic logic

1. Apply formal methods of symbolic propositional and predicate logic.
2. Recognize how formal tools of symbolic logic are used to model algorithms and real-life situations.
3. Use formal logic proofs and logical reasoning to solve problems such as puzzles.
4. Recognize the importance and limitations of predicate logic.

Proof techniques

1. Outline the basic structure of and give examples of each proof technique.
2. Discuss which type of proof is best for a given problem.
3. Relate the ideas of mathematical induction to recursion and recursively defined structures.
4. Identify the difference between mathematical and strong induction and give examples of the appropriate use of each.
5. Apply proof techniques to solve problems in computer science, including software engineering, program semantics, and algorithm analysis.

Basics of counting

1. Apply the sum and product rule for counting.
2. Explain the difference between combinations, C(n, r) and permutations, P(n, r).
3. Discuss Pascal's Identity for combinations and the Binomial Theorem.
4. Describe the concepts of initial condition, recurrence relation and the solution of recurrence relation.
5. Solve problems using arithmetic progressions, geometric progressions, and Fibonacci numbers.
6. Use the principle of inclusion-exclusion for counting.
7. Apply the pigeonhole principle.
8. Discuss examples of recurrence, including matrix multiplication, triangulation, and sorting.
9. Demonstrate the use of the Master Theorem to provide an instant asymptotic solution.
10. Distinguish among various cases of the Master Theorem.

Graphs and trees

1. Illustrate by example the basic terminology of graph theory, and some of the properties and special cases of each.
2. Examine different traversal methods for trees and graphs.
3. Model problems in computer science using graphs and trees.
4. Relate graphs and trees to data structures, algorithms, and counting.

Discrete probability

1. Calculate probabilities of events and expectations of random variables for elementary problems such as games of chance.
2. Differentiate between dependent and independent events.
3. Apply the Binomial Theorem to independent events and Bayes Theorem to dependent events.
4. Apply the tools of probability to solve problems such as the Monte Carlo method, the average case analysis of algorithms, and hashing.

Interpreting Descriptive Statistics

1. Distinguish between descriptive and inferential statistics and discuss the problems with using entire populations to obtain data.
2. Differentiate between numeric and qualitative data and review the graphing methods and statistics appropriate to each.
3. Use technology (e.g., Minitab, the TI-83 or Excel) to compute statistics and graph distributions.
4. Calculate and discuss the appropriate use of different measures of central tendency, variation, and position.
5. Discuss examples of the misuse of statistics.
6. Solve normal distribution problems, using z-scores and a table or technology.

SP: Social and Professional Issues Student Performance Objects

Methods and tools of analysis

1. Identify the premises and the conclusion in an ethical argument.
2. Recognize the basic logical fallacies in an ethical argument.
3. Discuss the ethical tradeoffs in a technical decision.

CS 105: Discrete Structures I

This course 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.

Prerequisites: Mathematical preparation sufficient to qualify for precalculus at the college level.
Corequisite: CS102

Syllabus:

- Introduction to logic and proofs: Direct proofs; proof by contradiction; mathematical induction
- Fundamental structures: Functions (surjections, injections, inverses, composition); relations (reflexivity, symmetry, transitivity, equivalence relations); sets (Venn diagrams, complements, Cartesian products, power sets); pigeonhole principle; cardinality and countability
- Boolean algebra: Boolean values; standard operations on Boolean values; de Morgan's laws
- Propositional logic: Logical connectives; truth tables; normal forms (conjunctive and disjunctive); validity
- Digital logic: Logic gates, flip-flops, counters; circuit minimization
- Descriptive statistics: methods of collecting data, frequency distribution graphs, measures of central tendency, variation, and position, and use of z-scores.
- Basics of counting: Counting arguments; pigeonhole principle; permutations and combinations; binomial coefficients

Knowledge Area	***Units Covered***	***Suggested Hours***
DS	Functions, relations, and sets	9
	Basic logic	5
	Proof techniques	4
	Basics of counting	9
	Interpreting descriptive statistics	9
AR	Digital logic and digital systems	3
SP	Methods and tools of analysis	1
	TOTAL:	**40**

Table 8: CS 105, Discrete Structures I

Notes:

The Discrete Structures (DS) material is divided into two courses. CS105 covers the first half of the material followed by CS106, which completes the topic coverage. Although the principal focus is discrete mathematics, the course is likely to be more successful if it highlights applications whose solutions require proof, logic, and counting.

CS 106: Discrete Structures II

This course continues the discussion of discrete mathematics introduced in CS 105. Topics in the second course include predicate logic, recurrence relations, graphs, trees, matrices, computational complexity, elementary computability, and discrete probability.

Prerequisite: CS 105
Corequisite: CS 103

Syllabus:

- Review of previous course
- Predicate logic: Universal and existential quantification; modus ponens and modus tollens; limitations of predicate logic
- Recurrence relations: Basic formulae; elementary solution techniques
- Graphs and trees: Fundamental definitions; simple algorithms; traversal strategies; proof techniques; spanning trees; applications
- Matrices: Basic properties; applications
- Computational complexity: Order analysis; standard complexity classes
- Elementary computability: Countability and uncountability; diagonalization proof to show uncountability of the reals; definition of the P and NP classes; simple demonstration of the halting problem
- Discrete probability: Finite probability spaces; conditional probability, independence, Bayes' rule; random events; random integer variables; mathematical expectation

Knowledge Area	***Units Covered***	***Suggested Hours***
DS	Basic logic	7
	Proof techniques	8
	Graphs and trees	4
	Discrete probability	6
AL	Basic algorithmic analysis	2
	Basic computability	3
	The complexity classes P and NP	2
Other	Matrices	3
	Topics of local interest	5
	TOTAL:	**40**

Table 9: CS 106, Discrete Structures II

Notes:

This implementation of the Discrete Structures area (DS) divides the material into two courses: CS105 and CS106. Like CS 105, CS 106 introduces mathematical topics in the context of applications that require those concepts as tools. For this course, likely applications include transportation network problems (such as the traveling salesperson problem) and resource allocation. Matrices have computing applications in many areas,

including inventory control, cost analysis, and data analysis. The unit on matrices introduces basic terminology, the operations of addition, scalar and matrix multiplication, the transpose and inverse, and 2x2 and 3x3 determinants.

Appendix D

ACM TYC Taxonomy of Learning Processes

Table 10 below is an adaptation of Bloom's Taxonomy (1956) by the ACM Two-Year College Education Committee used across all their curricular reports beginning with 1993.

Level of Taxonomy	Definition	Verbs to Help Design Activities
Factual Knowledge	**Recall information**	**Tell - list - define – name – recall - identify - remember – repeat – recognize**
Comprehension	**Understanding of communicated material or information**	**Transform - change - restate – describe - explain - interpret – summarize - discuss**
Applicative Knowledge	**Apply basic rules and conventions**	**Add – subtract – punctuate – edit – divide – multiply – diagram**
Procedural Knowledge	**Complete tasks using multi-step processes**	**Apply – investigate - produce**
Analysis	**Breaking down information into its parts**	**Analyze - dissect – distinguish - examine - compare - contrast – survey - categorize**
Synthesis	**Putting together ideas into a new or unique product**	**Create – invent – compose – construct - design - produce – modify**
Evaluation	**Judging the value of materials or ideas based on set standards or criteria**	**Judge - decide – justify – evaluate - critique - debate – verify – recommend**
Higher-Order Thinking	**Apply analysis, syntheses and evaluation processes to solve complex problems**	**Evaluate - create – conduct – analyze**
Attitudes and Values	**Express feelings, opinions, personal beliefs regarding people, objects and events**	**Respect – demonstrate – express**
Social Behaviors	**Learned behavior that conforms to acceptable social standards**	**Perform – communicate**
Motor Skills	**Physical coordination, strength, control, skills related to physical tasks**	**Demonstrate - run – dribble - move - show**

Table 10: ACM TYC Taxonomy of Learning Processes