Computer Science Curricula 2013

Strawman Draft

(February 2012)

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

CS2013 Steering Committee

ACM Delegation

Mehran Sahami, *Chair* (Stanford University)

Andrea Danyluk (Williams College)

Sally Fincher (University of Kent)

Kathleen Fisher (Tufts University)

Dan Grossman (University of Washington)

Beth Hawthorne (Union County College)

Randy Katz (UC Berkeley)

Rich LeBlanc (Seattle University)

Dave Reed (Creighton University)

IEEE-CS Delegation

Steve Roach, *Chair* (Univ. of Texas, El Paso)

Ernesto Cuadros-Vargas (Univ. Catolica San Pablo)

Ronald Dodge (US Military Academy)

Robert France (Colorado State University)

Amruth Kumar (Ramapo Coll. of New Jersey)

Brian Robinson (ABB Corporation)

Remzi Seker (Univ. of Arkansas, Little Rock)

Alfred Thompson (Microsoft)

Table of Contents

Chapter 1: Introduction

Continuing a process that began over 40 years ago with the publication of Curriculum 68 [1], the major professional societies in computing—ACM and IEEE-Computer Society—have sponsored efforts to establish international curricular guidelines for undergraduate programs in computing on roughly a 10-year cycle. As the field of computing has grown and diversified, so too have the curricular recommendations, and there are now curricular volumes for Computer Engineering, Information Systems, Information Technology, and Software Engineering in addition to Computer Science [3]. These volumes are updated regularly with the aim of keeping computing curricula modern and relevant. The last complete Computer Science curricular volume was released in 2001 (CC2001) [2], and an interim review effort concluded in 2008 (CS2008) [4].

This volume, Computer Science Curricula 2013 (CS2013), represents a comprehensive revision. CS2013 redefines the knowledge units in CS, rethinking the essentials necessary for a Computer Science curriculum. It also seeks to identify exemplars of actual courses and programs to provide concrete guidance on curricular structure and development in a variety of institutional contexts.

The development of curricular guidelines for Computer Science is particularly challenging given the rapid evolution and expansion of the field: material dates fast. Moreover, the growing diversity of topics in Computer Science and the increasing integration of computing with other disciplines create additional challenges. Balancing topical growth with the need to keep recommendations realistic and implementable in the context of undergraduate education is particularly difficult. As a result, it is important to engage the broader computer science education community in a dialog to better understand new opportunities, local needs, and to identify successful models of computing curriculum – whether established or novel. One aim of this Strawman report is to provide the basis for such engagement, by providing an early draft of the CS2013 volume that can be scrutinized by members of the computing community with the goal of augmenting and refining the final report.

Charter

The ACM and IEEE-Computer Society chartered the CS2013 effort with the following directive:

> *To review the Joint ACM and IEEE-CS Computer Science volume of Computing Curricula 2001 and the accompanying interim review CS 2008, and develop a revised and enhanced version for the year 2013 that will match the latest developments in the discipline and have lasting impact.*
>
> *The CS2013 task force will seek input from a diverse audience with the goal of broadening participation in computer science. The report will seek to be international in scope and offer curricular and pedagogical guidance applicable to a wide range of institutions. The process of producing the final report will include multiple opportunities for public consultation and scrutiny.*

Consequently, the CS2013 task force welcomes review of, and comment on, this draft report.

High-level Themes

In developing CS2013, several high-level themes provided an overarching guide for this volume. These themes, which embody and reflect the CS2013 Principles (described in detail in another section of this volume) are:

- *The "Big Tent" view of CS*. As CS expands to include more cross-disciplinary work and new programs of the form "Computational Biology," "Computational Engineering," and "Computational X" are developed, it is important to embrace an outward-looking view that sees CS as a discipline actively seeking to work with and integrate into other disciplines.
- *Managing the size of the curriculum.* Although the field of Computer Science continues to grow unabated, it is not feasible to proportionately expand the size of the curriculum. As a result, CS2013 seeks to re-evaluate the essential topics in computing to make room for new topics without requiring more total instructional hours than the CS2008 guidelines. At the same time, the circumscription of curriculum size promotes more flexible models for curricula without losing the essence of a rigorous CS education.
- *Actual course exemplars.* CS2001 took on the significant challenge of providing descriptions of six *curriculum models* and forty-seven possible *course descriptions* variously incorporating the knowledge units as defined in that report. While this effort was valiant, in retrospect such course guidance did not seem to have much impact on actual course design. CS2013 plans to take a different approach: to identify and describe existing successful courses and curricula to show how relevant knowledge units are addressed and incorporated in actual programs.

- *Institutional needs.* CS2013 aims to be applicable in a broad range of geographic and cultural contexts, understanding that curricula exist within specific institutional needs, goals, and resource constraints. As a result, CS2013 allows for explicit flexibility in curricular structure through a tiered set of core topics, where a small set of Core-Tier 1 topics are considered essential for all CS programs, but individual programs choose their coverage of Core-Tier 2 topics. This tiered structure is described in more detail in Chapter 4 of this report.

Knowledge Areas

The CS2013 Body of Knowledge is organized into a set of 18 Knowledge Areas (KAs), corresponding to topical areas of study in computing. The Knowledge Areas are:

- AL - Algorithms and Complexity
- AR - Architecture and Organization
- CN - Computational Science
- DS - Discrete Structures
- GV - Graphics and Visual Computing
- HC - Human-Computer Interaction
- IAS - Information Assurance and Security
- IM - Information Management
- IS - Intelligent Systems
- NC - Networking and Communications
- OS - Operating Systems
- PBD - Platform-based Development
- PD - Parallel and Distributed Computing
- PL - Programming Languages
- SDF - Software Development Fundamentals
- SE - Software Engineering
- SF - Systems Fundamentals
- SP - Social and Professional Issues

Many of these Knowledge Areas are derived from CC2001/CS2008 but have been revised—in some cases quite significantly—in CS2013; others are new. There are three major causes of KA change: the reorganization of existing KAs, the development of cross-cutting KAs, and the creation of entirely new KAs. Reorganized KAs are a refactoring of existing topics to better reflect coherent units of knowledge as the field of Computer Science has evolved. For example, Software Development Fundamentals is a significant reorganization of the previous Programming Fundamentals KA. Cross-cutting KAs are a refactoring of existing KAs that extract and integrates cross-cutting foundational topics into their own KA rather than duplicating them across many others. Examples include SF-System Fundamentals and IAS-Information Assurance and Security. Finally, new KAs reflect emerging topics in CS that have become sufficiently prevalent to be included in the volume. PBD-Platform-based Development is an example of such a KA. Chapter 5 contains a more comprehensive overview of these changes.

Previous Input

To lay the groundwork for CS2013, we conducted a survey of the usage of the CC2001 and CS2008 volumes. The survey was sent to approximately 1500 Computer Science (and related discipline) Department Chairs and Directors of Undergraduate Studies in the United States and an additional 2000 Department Chairs internationally. We received 201 responses, representing a wide range of institutions (self-identified):

- research-oriented universities (55%)
- teaching-oriented universities (17.5%)
- undergraduate-only colleges (22.5%)
- community colleges (5%)

The institutions also varied considerably in size, with the following distribution:

- less than 1,000 students (6.5%)
- 1,000 to 5,000 students (30%)
- 5,000 to 10,000 students (19%)
- more than 10,000 students (44.5%)

In examining the *usage* of the CC2001/CS2008 reports, survey respondents reported that the Body of Knowledge (i.e., the outline of topics that should appear in undergraduate Computer Science curricula) was the most used aspect. When questioned about new topical areas that should be added to the Body of Knowledge, survey respondents indicated a strong need to add the topics of *Security* as well as *Parallel and Distributed Computing*. Indeed, feedback during the CS2008 review had also indicated the importance of these two areas, but the CS2008 steering committee had felt that creating new KAs was beyond their purview and deferred the development of those areas to the next full curricular report. CS2013 includes these two new KAs (among others): *Information Assurance and Security*, and *Parallel and Distributed Computing*.

Coming Attractions in CS2013

The final version of the CS2013 volume is, naturally enough, scheduled for release in 2013. Hence, this Strawman draft is—by design—incomplete. Not only will the final report include revisions of the Body of Knowledge presented here, based on community feedback, it will also include several sections which do not yet exist. Here we provide a timeline for CS2013 efforts and outline some of the "coming attractions" (i.e., additional sections) that are planned for inclusion in future drafts.

Timeline

The 2013 curricular guidelines will comprise several sorts of materials: the Body of Knowledge, Exemplars of Curricula and Courses, Professional Practice, and Institutional Challenges. These are being developed in offset phases, starting with the Body of Knowledge.

A summary of the CS2013 timeline is as follows:

Fall 2010:	CS2013 chartered and effort begins
February 2011:	CS2013 Principles outlined and Body of Knowledge revision begins
February 2012:	CS2013 Strawman report released Includes: Body of Knowledge, Characteristics of Graduates
July 15, 2012:	Comment period for Strawman draft closes
February 2013:	CS2013 Ironman report planned for release Includes: Body of Knowledge, Characteristics of Graduates, Curricula and Course Exemplars, Professional Practice, Institutional Challenges
June 2013:	Comment period for Ironman draft closes
Summer 2013:	CS2013 Final report planned for release

Exemplars of Curricula and Courses

Perhaps the most significant section of the CS2013 final report that is not included in the Strawman draft is the presentation of actual curricula and courses that embody the topics in the CS2013 Body of Knowledge. The CS2013 Ironman draft will include examples used in practice—from a variety of universities and colleges—to illustrate how topics in the Knowledge Areas may be covered and combined in diverse ways.

Importantly, we believe that the identification of such exemplary courses and curricula provides a tremendous opportunity for further community involvement in the development of the CS2013 volume. We invite members of the computing community to contribute courses and curricula from their own institutions (or other institutions that they may be familiar with). Those interested in potentially mapping courses/curricula to the CS2013 Body of Knowledge are encouraged to contact members of the CS2013 steering committee for more details.

Professional Practice

The education that undergraduates in Computer Science receive must adequately prepare them for the workforce in a more holistic way than simply conveying technical facts. Indeed, “soft skills” (such as teamwork and communication) and personal attributes (such as identification of opportunity and risk) play a critical role in the workplace. Successfully applying technical knowledge in practice often requires an ability to tolerate ambiguity and work well with others from different backgrounds and disciplines. These overarching considerations are important for promoting successful professional practice in a variety of career paths. We will include suggestions for, and examples of, ways in which curricula encourage the development of such skills, including professional competencies and entrepreneurship, as part of an undergraduate Computer Science program in the CS2013 Ironman draft.

Institutional Challenges

CS departments and programs often face institutional challenges in implementing a curriculum: they may have too few faculty to cover all the knowledge areas, insufficient number of students for a full program, and/or inadequate institutional resource for professional development. This section will identify such challenges and provide suggestions for their amelioration.

Opportunities for Involvement

We believe it is essential for endeavours of this kind to engage the broad computing community to review and critique successive drafts. To this end, the development of this Strawman report has already benefited from the input of more than 100 contributors beyond the steering committee. We welcome further community engagement on this effort in multiple ways, including (but not limited to):

- Comments on the Strawman draft, especially with respect to the Body of Knowledge.
- Contribution of exemplar courses/curricula that are mapped against the Body of Knowledge.
- Descriptions of pedagogic approaches and instructional designs (both time-tested and novel) that address professional practice within undergraduate curricula.
- Sharing of institutional challenges, and solutions to them.

Comments on all aspects of this report are welcome and encouraged via the CS2013 website:

`http://cs2013.org`

References

[1] ACM Curriculum Committee on Computer Science. 1968. Curriculum 68: Recommendations for Academic Programs in Computer Science. *Comm. ACM* 11, 3 (Mar. 1968), 151-197.

[2] ACM/IEEE-CS Joint Task Force on Computing Curricula. 2001. ACM/IEEE Computing Curricula 2001 Final Report. http://www.acm.org/sigcse/cc2001.

[3] ACM/IEEE-CS Joint Task Force for Computer Curricula 2005. Computing Curricula 2005: An Overview Report. http://www.acm.org/education/curric_vols/CC2005-March06Final.pdf

[4] ACM/IEEE-CS Joint Interim Review Task Force. 2008. Computer Science Curriculum 2008: An Interim Revision of CS 2001, Report from the Interim Review Task Force. http://www.acm.org/education/curricula/ComputerScience2008.pdf

Acknowledgments

The CS2013 Strawman report has benefited from the input of many individuals, including: Alex Aiken (Stanford University), Ross Anderson (Cambridge University), Florence Appel (Saint Xavier University), Helen Armstrong (Curtin university), Colin Armstrong (Curtin university), Krste Asanovic (UC Berkeley), Radu F. Babiceanu (University of Arkansas at Little Rock), Mike Barker (Massachusetts Institute of Technology), Michael Barker (Nara Institute of Science and Technology), Paul Beame (University of Washington), Robert Beck (VIllanova University), Matt Bishop (University of California, Davis), Alan Blackwell (Cambridge University), Don Blaheta (Longwood University), Olivier Bonaventure (Universite Catholique de Louvain), Roger Boyle (University of Leeds), Clay Breshears (Intel), Bo Brinkman (Miami University), David Broman (Linkoping University), Kim Bruce (Pomona College), Jonathan Buss (University of Waterloo), Netiva Caftori (Northeastern Illinois University, Chicago), Paul Cairns (University of York), Alison Clear (Christchurch Polytechnic Institute of Technology), Curt Clifton (Rose-Hulman and The Omni Group), Yvonne Cody (University of Victoria), Tony Cowling (University of Shefffield), Joyce Currie-Little (Towson University), Ron Cytron (Washington University in St. Louis), Melissa Dark (Purdue University), Janet Davis (Grinnell College), Marie DesJardins (University of Maryland, Baltimore County), Zachary Dodds (Harvey Mudd College), Paul Dourish (University of California, Irvine), Lynette Drevin (North-West Universit), Scot Drysdale (Dartmouth College), Kathi Fisler (Worcester Polytechnic Institute), Susan Fox (Macalester College), Edward Fox (Virginia Tech), Eric Freudenthal (University of Texas El Paso), Stephen Freund (Williams College), Lynn Futcher (Nelson Mandela Metropolitan University), Greg Gagne (Wesminister College), Dan Garcia (UC Berkeley), Judy Gersting (Indiana University-Purdue University Indianapolis), Yolanda Gil (University of Southern California), Michael Gleicher (University Wisconsin, Madison), Frances Grodzinsky (Sacred Heart University), Anshul Gupta (IBM), Mark Guzdial (Georgia Tech), Brian Hay (University of Alaska, Fairbanks), Brian Henderson-Sellers (University of Technology, Sydney), Matthew Hertz (Canisius College), Tom Hilburn (Embry-Riddle Aeronautical University), Tony Hosking (Purdue University), Johan Jeuring (Utrecht University), Yiming Ji (University of South Carolina Beaufort), Maggie Johnson (Google), Matt Jones (Swansea University), Frans Kaashoek (MIT), Lisa Kaczmarczyk (ACM Education Council), Jennifer Kay (Rowan

University), Scott Klemmer (Stanford University), Jim Kurose (University of Massachusetts, Amherst), Doug Lea (SUNY Oswego), Terry Linkletter (Central Washington University), David Lubke (NVIDIA), Bill Manaris (College of Charleston), Samuel Mann (Otago Polytechnic), C. Diane Martin (George Washington University), Andrew McGettrick (University of Strathclyde), Morgan Mcguire (Williams College), Keith Miller (University of Illinois at Springfield), Narayan Murthy (Pace University), Kara Nance (University of Alaska, Fairbanks), Todd Neller (Gettysburg College), Reece Newman (Sinclair Community College), Christine Nickell (Information Assurance Center for Computer Network Operations, CyberSecurity, and Information Assurance), James Noble (Victoria University of Wellington), Peter Norvig (Google), Joseph O'Rourke (Smith College), Jens Palsberg (UCLA), Robert Panoff (Shodor.org), Sushil Prasad (Georgia State University), Michael Quinn (Seattle University), Matt Ratto (University of Toronto), Penny Rheingans (U. Maryland Baltimore County), Carols Rieder (Lucerne University of Applied Sciences), Eric Roberts (Stanford University), Arny Rosenberg (Northeastern and Colorado State University), Ingrid Russell (University of Hartford), Dino Schweitzer (United States Air Force Academy), Michael Scott (University of Rochester), Robert Sedgewick (Princeton University), Helen Sharp (Open Univeristy), Robert Sloan (University of Illinois, Chicago), Ann Sobel (Miami University), Carol Spradling (Northwest Missouri State University), Michelle Strout (Colorado State University), Alan Sussman (University of Maryland, College Park), Blair Taylor (Towson University), Simon Thompson (University of Kent), Johan Vanniekerk (Nelson Mandela Metropolitan University), Christoph von Praun (Georg-Simon-Ohm Hochschule Nürnberg), Rossouw Von Solms (Nelson Mandela Metropolitan University), John Wawrzynek (UC Berkeley), Charles Weems (Univ. of Massachusettes, Amherst), David Wetherall (University of Washington), Michael Wrinn (Intel)

Additionally, review of various portions of the Strawman report took part in several venues, including: the 42nd ACM Technical Symposium of the Special Interest Group on Computer Science Education (SIGCSE-11), the 24th IEEE-CS Conference on Software Engineering Education and Training (CSEET-11), the 2011 IEEE Frontiers in Education Conference (FIE-11), the 2011 Federated Computing Research Conference (FCRC-11), the 2nd Symposium on Educational Advances in Artificial Intelligence (EAAI-11), the Conference of ACM Special Interest Group on Data Communication 2011 (SIGCOMM-11), the 2011 IEEE International Joint Conference on Computer, Information, and Systems Sciences and Engineering (CISSE-11), the 2011 Systems, Programming, Languages and Applications: Software for Humanity Conference (SPLASH-11), the 15th Colloquium for Information Systems Security Education, the 2011 National Centers of Academic Excellence in IA Education (CAE/IAE) Principles meeting, and the 7th IFIP TC 11.8 World Conference on Information Security Education (WISE).

Several more conference special sessions to review and comment on drafts of CS2013 are planned for the coming year, including 43rd ACM Technical Symposium of the Special Interest Group on Computer Science Education (SIGCSE-12), the Special Session of the Special Interest Group on Computers and Society at SIGCSE-12, Computer Research Association Snowbird Conference 2012, and the 2012 IEEE Frontiers in Education Conference (FIE-12), among others.

A number of organizations also provided valuable feedback to the CS2013 Strawman effort, including: the ACM Education Board and Council, the IEEE-CS Educational Activities Board, the ACM SIGPLAN Education Board, the ACM Special Interest Group Computers and Society, and the NSF/IEEE-TCPP Curriculum Initiative on Parallel and Distributed Computing Committee

Chapter 2: Principles

Early in its work, the 2013 Steering Committee agreed upon a set of principles to guide the development of this volume. The principles adopted for CS2013 overlap significantly with the principles adopted for previous curricular efforts, most notably CC2001 and CS2008. As with previous ACM/IEEE curricula volumes, there are a variety of constituencies for CS2013, including individual faculty members and instructors at a wide range of colleges, universities, and technical schools on any of six continents; CS programs and the departments, colleges, and institutions where they are housed; accreditation and certification boards; authors; and researchers. Other constituencies include pre-college preparatory schools and advanced placement curricula as well as graduate programs in computer science.

The principles were developed in consideration of these constituencies, as well as issues related to student outcomes, development of curricula, and the review process. The order of presentation is not intended to imply relative importance.

1. *Computer Science curricula should be designed to provide students with the flexibility to work across many disciplines.* Computing is a broad field that connects to and draws from many disciplines, including mathematics, electrical and systems engineering, psychology, statistics, fine arts, linguistics, and physical and life sciences. Computer Science students should develop the flexibility to work across disciplines.

2. *Computer Science curricula should be designed to prepare graduates for a variety of professions, attracting the full range of talent to the field.* Computer Science impacts nearly every modern endeavour. CS2013 takes a broad view of the field that includes topics such as “computational-x” (e.g., computational finance or computational chemistry) and “x-informatics” (e.g., eco-informatics or bio-informatics). Well-rounded CS graduates will have a balance of theory and application, as described in Chapter 3: Characteristics of Graduates.

3. *CS2013 should provide guidance for the expected level of mastery of topics by graduates*. It should suggest outcomes indicating the intended level of mastery and provide exemplars of fielded curricula covering topics in the Body of Knowledge.

4. *CS 2013 must provide realistic, adoptable recommendations that provide guidance and flexibility, allowing curricular designs that are innovative and track recent developments in the field.* The guidelines are intended to provide clear, implementable goals, while also providing the flexibility that programs need in order to respond to a rapidly changing field. CS2013 is intended as guidance, not as a minimal standard against which to evaluate a program.

5. *The CS2013 guidelines must be relevant to a variety of institutions.* Given the wide range of institutions and programs (including 2-year, 3-year, and 4-year programs; liberal arts, technological, and research institutions; and institutions of every size), it is neither possible nor desirable for these guidelines to dictate curricula for computing. Individual programs will need to evaluate their constraints and environments to construct curricula.

6. *The size of the essential knowledge must be managed*. While the range of relevant topics has expanded, the size of undergraduate curricula has not. Thus, CS2013 must carefully choose among topics and recommend the essential elements.

7. *Computer Science curricula should be designed to prepare graduates to succeed in a rapidly changing field.* Computer Science is rapidly changing and will continue to change for the foreseeable future. Curricula must prepare students for lifelong learning and must include professional practice (e.g. communication skills, teamwork, ethics) as components of the undergraduate experience. Computer science students must learn to integrate theory and practice, to recognize the importance of abstraction, and to appreciate the value of good engineering design.

8. *CS2013 should identify the fundamental skills and knowledge that all computer science graduates should possess while providing the greatest flexibility in selecting topics.* To this end, we have introduced three levels of knowledge description: Tier-1 Core, Tier-2 Core, and Elective. For a full discussion of Tier-1 Core, Tier-2 Core, and Elective, see Chapter 4: Completing the Curriculum.

9. *CS2013 should provide the greatest flexibility in organizing topics into courses and curricula.* Knowledge areas are not intended to describe specific courses. There are many

novel, interesting, and effective ways to combine topics from the Body of Knowledge into courses.

10. *The development and review of CS2013 must be broadly based.* The CS2013 Task Force must include participation from many different constituencies including industry, government, and the full range of higher education institutions involved in computer science education. It must take into account relevant feedback from these constituencies.

Chapter 3: Characteristics of Graduates

Graduates of Computer Science programs should have fundamental competency in the areas described by the Body of Knowledge (see Chapter 5), particularly the core topics contained there. However, there are also competences that graduates of CS programs should have that are not explicitly listed in the Body of Knowledge. Professionals in the field typically embody a characteristic style of thinking and problem solving, a style that emerges from the experiences obtained through study of the field and professional practice. Below, we describe the characteristics that we believe should be met at least at an elementary level by graduates of computer science programs. These characteristics will enable their success in the field and further professional development. Some of these characteristics and skills also apply to other fields. They are included here because the development of these skills and characteristics must be explicitly addressed and encouraged by Computer Science programs.

This list is based on a similar list in CC2001 and CS2008. The substantial changes that led to this new version were influenced by responses to a survey conducted by the CS2013 Steering Committee.

At a broad level, the expected characteristics of computer science graduates include the following:

Technical understanding of Computer Science

Graduates should have a mastery of computer science as described by the core of the Body of Knowledge.

Familiarity with common themes and principles

Graduates need understanding of a number of recurring themes, such as abstraction, complexity, and evolutionary change, and a set of general principles, such as sharing a common resource, security, and concurrency. Graduates should recognize that these themes and principles have broad application to the field of computer science and should not consider them as relevant only to the domains in which they were introduced.

Appreciation of the interplay between theory and practice

A fundamental aspect of computer science is understanding the interplay between theory and practice and the essential links between them. Graduates of a computer science program need to understand how theory and practice influence each other.

System-level perspective

Graduates of a computer science program need to think at multiple levels of detail and abstraction. This understanding should transcend the implementation details of the various components to encompass an appreciation for the structure of computer systems and the processes involved in their construction and analysis. They need to recognize the context in which a computer system may function, including its interactions with people and the physical world.

Problem solving skills

Graduates need to understand how to apply the knowledge they have gained to solve real problems, not just write code and move bits. They should also realize that there are multiple solutions to a given problem and that selecting among them is not a purely technical activity, as these solutions will have a real impact on people's lives. Graduates also should be able to communicate their solution to others, including why and how a solution solves the problem and what assumptions were made.

Project experience

To ensure that graduates can successfully apply the knowledge they have gained, all graduates of computer science programs should have been involved in at least one substantial project. In most cases, this experience will be a software development project, but other experiences are also appropriate in particular circumstances. Such projects should challenge students by being integrative, requiring evaluation of potential solutions, and requiring work on a larger scale than typical course projects. Students should have opportunities to develop their interpersonal communication skills as part of their project experience.

Commitment to life-long learning

Graduates of a computer science program should realize that the computing field advances at a rapid pace. Specific languages and technology platforms change over time. Therefore, graduates need to realize that they must continue to learn and adapt their skills throughout their careers. To develop this ability, students should be exposed to multiple programming languages, tools, and technologies as well as the fundamental underlying principles throughout their education.

Commitment to professional responsibility

Graduates should recognize the social, legal, ethical and cultural issues involved in the deployment and use of computer technology. They should respond to these issues from an informed perspective, guided by personal and professional principles. They must further recognize that social, legal, and ethical standards vary internationally.

Communication and organizational skills

Graduates should have the ability to make succinct presentations to a range of audiences about technical problems and their solutions. This may involve face-to-face, written, or electronic communication. They should be prepared to work effectively as members of teams. Graduates should be able to manage their own learning and development, including managing time, priorities, and progress.

Awareness of the broad applicability of computing

Platforms range from embedded micro-sensors to high-performance clusters and distributed clouds. Computer applications impact nearly every aspect of modern life. Graduates should understand the full range of opportunities available in computing.

Appreciation of domain-specific knowledge

Graduates should understand that computing interacts with many different domains. Solutions to many problems require both computing skills and domain knowledge. Therefore, graduates need to be able to communicate with, and learn from, experts from different domains throughout their careers.

Chapter 4: Constructing a Complete Curriculum

This chapter provides high-level guidelines on how to use the Body of Knowledge to create an institution's undergraduate curriculum in computer science. It does not propose a particular set of courses or curriculum structure -- that is the role of the (forthcoming) course/curriculum exemplars. Rather, this chapter emphasizes the flexibility that the Body of Knowledge allows in adapting curricula to institutional needs and the continual evolution of the field. In computer-science terms, one can view the Body of Knowledge as a specification of content to cover and a curriculum as an implementation. A large variety of curricula can meet the specification.

The following points are elaborated:

- Knowledge Areas are not intended to be in one-to-one correspondence with particular courses in a curriculum: We expect curricula will have courses incorporating topics from multiple Knowledge Areas.
- Topics are identified as either "core" or "elective" with the core further subdivided into "tier-1" and "tier-2."
    - A curriculum should include all topics in the tier-1 core and ensure that all students cover this material.
    - A curriculum should include all or almost all topics in the tier-2 core and ensure that all students cover the vast majority of this material.
    - A curriculum should include significant elective material: Covering only "core" topics is insufficient for a complete curriculum.
- Because it is a hierarchical outline, the Body of Knowledge under-emphasizes some key issues that must be considered when constructing a curriculum.

Knowledge Areas are Not Necessarily Courses (and Important Examples Thereof)

It is naturally tempting to associate each Knowledge Area with a course. We explicitly discourage this practice in general, even though many curricula will have some courses containing material from only one Knowledge Area or, conversely, all the material from one Knowledge Area in one course. We view the hierarchical structure of the Body of Knowledge as a useful way to group related information, not as a stricture for organizing material into courses. Beyond this general flexibility, in several places we expect many curricula to integrate material from multiple Knowledge Areas, in particular:

- *Introductory courses:* There are diverse successful approaches to introductory courses in computer science. Many focus on the topics in Software Development Fundamentals together with a subset of the topics in Programming Languages or Software Engineering, while leaving most of the topics in these other Knowledge Areas to advanced courses. But *which* topics from other Knowledge Areas are covered in introductory courses can vary. Some courses use object-oriented programming, others functional programming, others platform-based development (thereby covering topics in the Platform-Based Development Knowledge Area), etc. Conversely, there is no requirement that all Software Development Fundamentals be covered in a first or second course, though in practice most topics will usually be covered in these early courses.

- *Systems courses:* The topics in the Systems Fundamentals Knowledge Area can be covered in courses designed to cover general systems principles or in courses devoted to particular systems areas such as computer architecture, operating systems, networking, or distributed systems. For example, an Operating Systems course might spend considerable time on topics of more general use, such as low-level programming, concurrency and synchronization, performance measurement, or computer security. Such courses may draw on material in several Knowledge Areas. Certain fundamental systems topics like latency or parallelism will likely arise in many places in a curriculum. While it is important that such topics do arise, preferably in multiple settings, the Body of Knowledge does not specify the particular settings in which to teach such topics.

- *Parallel computing:* Among the many changes to the Body of Knowledge compared to previous reports is a new Knowledge Area in Parallel and Distributed Computing. An alternative structure for the Body of Knowledge would place relevant topics in other Knowledge Areas: parallel algorithms with algorithms, programming constructs in software-development focused areas, multi-core design with computer architecture, and so forth. We chose instead to provide guidance on the essential parallelism topics in one place. Some, but not all, curricula will likely have courses dedicated to parallelism, at least in the near term.

Tier-1 Core, Tier-2 Core, Elective: What These Terms Mean, What is Required

As described at the beginning of this chapter, computer science curricula should cover all of the core tier-1 topics, all or almost all of the core tier-2 topics, and significant depth in many of the elective topics (i.e., the core is not sufficient for an undergraduate degree in computer science). Here we provide additional perspective on what “tier-1 core,” “tier-2 core”, and “elective” mean, including motivation for these distinctions.

Motivation for subdividing the core: Earlier versions of the ACM/IEEE Computer Science Curricula had only “core” and “elective” with every topic in the former being required. We departed from this strict interpretation of “everything in the core must be taught to every student” for these reasons:

- It did not sufficiently reflect reality: Many strong computer science curricula were missing at least one hour of core material. It is misleading to suggest that such curricula are outside the definition of an undergraduate degree in computer science.

- As the field has grown, there is ever-increasing pressure to grow the core and allow students to specialize in areas of interest. Doing so simply becomes impossible within the short time-frame of an undergraduate degree. Providing some flexibility on coverage of core topics enables curricula and students to specialize if they choose to do so.

Conversely, we could have allowed for *any* core topic to be skipped provided that the vast majority was part of every student’s education. By retaining a smaller tier-1 core of required material, we provide additional guidance and structure for curriculum designers. In the tier-1 core are the topics that are fundamental to the structure of any computer-science program.

On the meaning of tier-1: A tier-1 topic should be a required part of every computer-science curriculum for every student. This is not to say that tier-2 or even elective topics should not be, but the tier-1 topics are those with widespread consensus for inclusion. Moreover, at least preliminary treatment of most of these topics typically comes in the first two years of a curriculum, precisely because so much of the field relies on these topics. However, introductory courses need not cover all tier-1 material and will usually draw on tier-2 and elective material as well.

On the meaning of tier-2: Tier-2 topics are generally essential in an undergraduate computer-science degree. Requiring the vast majority of them is a *minimum* expectation, and we encourage institutions to cover all of them for every student. That said, computer science programs can allow students to focus in certain areas in which some tier-2 topics are not required. We also acknowledge that resource constraints, such as a small number of faculty or institutional limits on degree requirements, may make it prohibitively difficult to cover every topic in the core while still providing advanced elective material. **A computer-science curriculum should aim to cover 90-100% of the tier-2 topics for every student, with 80% considered as a minimum.**

There is no expectation that tier-1 topics necessarily precede tier-2 topics in a curriculum. In particular, we expect introductory courses will draw on both tier-1 and tier-2 (and possibly elective) material and that some core material will be delayed until later courses.

On the meaning of elective: A program covering only core material would provide insufficient breadth and depth in computer science, but most programs will not cover all the elective material in the Body of Knowledge and certainly few, if any, students will cover all of it within an undergraduate program. Conversely, the Body of Knowledge is by no means exhaustive, and advanced courses may often go beyond the topics and learning outcomes contained in it. Nonetheless, the Body of Knowledge provides a useful guide on material appropriate for a computer-science undergraduate degree, and all students of computer science should deepen their understanding in multiple areas via the elective topics.

A curriculum may well require material designated elective in the Body of Knowledge. Many curricula, especially those with a particular focus, will require some elective topics, by virtue of them being covered in required courses.

The size of the core: The size of the core (tier-1 plus tier-2) is a few hours larger than in previous versions of the computer-science curriculum, but this is counterbalanced by our more flexible treatment of the core. As a result, we are not increasing the number of required courses a curriculum should need. Indeed, a curriculum covering 90% of the tier-2 hours would have the same number of core hours as a curriculum covering the core in the CS2008 volume, and a curriculum covering 80% of the tier-2 hours would have fewer core hours than even a curriculum covering the core in the CC2001 volume (the core grew from 2001 to 2008).

A note on balance: Computer science is an elegant interplay of theory, software, hardware, and applications. The core in general and the tier-1 core in particular, when viewed in isolation, may seem to focus on programming, discrete structures, and algorithms. This focus results from the fact that these topics typically come early in a curriculum so that advanced courses can use them as pre-requisites. Essential experience with systems and applications can be achieved in more disparate ways using elective material in the Body of Knowledge. Because all curricula will include appropriate elective material, an overall curriculum can and should achieve an appropriate balance.

Further Considerations

As useful as the Body of Knowledge is, it is important to complement it with a thoughtful understanding of cross-cutting themes in a curriculum, the "big ideas" of computer science. In designing a curriculum, it is also valuable to identify curriculum-wide objectives, for which the Principles and the Characteristics of Graduates chapters of this volume should prove useful.

In the last few years, two on-going trends have had deep effects on many curricula. First, the continuing growth of computer science has led to many programs organizing their curricula to allow for *intradisciplinary* specialization (using terms such as threads, tracks, vectors, etc.). Second, the importance of computing to almost every other field has increasingly led to the creation of *interdisciplinary* programs (joint majors, double majors, etc.) and incorporating interdisciplinary material into computer-science programs. We applaud both trends and believe a flexible Body of Knowledge, including a flexible core, support them. Conversely, such specialization is not required: Many programs will continue to offer a broad yet thorough coverage of computer science as a distinct and coherent discipline.

Chapter 5: Introduction to the Body of Knowledge

Process for Updating the Body of Knowledge

The CS2013 Steering Committee constituted a subcommittee for each KA, chaired by a member of the Steering Committee, and initially including at least two other members of the Steering Committee. Individual subcommittee Chairs then invited expert members (outside the CS2013 Steering Committee) to join the work of defining and reviewing each KA; drafts of KAs were also presented in various conference panel and special session presentations. The KA subcommittee Chairs (as members of the CS2013 Steering Committee) worked to resolve conflicts, eliminate redundancies and appropriately categorize and cross-reference topics between the various KAs. This year-long process ultimately converged to the draft version of the Body of Knowledge presented here.

As noted in the introduction to this report, we are soliciting continued community feedback which will be considered and incorporated into future drafts of the CS2013 report.

The CS2013 Body of Knowledge is presented as a set of Knowledge Areas (KAs), organized on topical themes rather than by course boundaries. Each KA is further organized into a set of Knowledge Units (KUs), which are summarized in a table at the head of each KA section. We expect that the topics within the KAs will be organized into courses in different ways at different institutions.

Here, we provide background for understanding how to read the Body of Knowledge, and we give an overview of the number of core hours in each KA. We also highlight the KAs that have significant cross-topic components and those that are new to this volume. Chapter 4 presents essential background on how the Body of Knowledge translates into actual curricula.

Overview of New Knowledge Areas

While computer science encompasses technologies that change rapidly over time, it is defined by essential concepts, perspectives, and methodologies that are constant. As a result, much of the core Body of Knowledge remains unchanged from earlier curricular volumes. However, new developments in computing technology and pedagogy mean that some aspects of the core evolve over time, and some of the previous structures and organization may no longer be appropriate for describing the discipline. As a result, CS2013 has modified the organization of the curriculum in various ways, adding some new KAs and restructuring others. We highlight these changes in the remainder of this section.

IAS-Information Assurance and Security

IAS is a new KA in recognition of the world's reliance on information technology and its critical role in computer science education. IAS as a domain is the set of controls and processes, both technical and policy, intended to protect and defend information and information systems. IAS draws together topics that are pervasive throughout other KAs. Topics germane to *only* IAS are presented in depth in this KA, whereas other topics are noted and cross referenced to the KAs that contain them. As such, this KA is prefaced with a detailed table of cross-references to other KAs.

NC-Networking and Communication

CC2001 introduced a KA entitled "Net-Centric Computing" which encompassed a combination of topics including traditional networking, web development, and network security. Given the growth and divergence in these topics since the last report, we renamed and refactored this KA to focus specifically on topics in networking and communication. Discussions of web applications and mobile device development are now covered in the new PBD-Platform-Based Development KA. Security is covered in the new IAS-Information Assurance and Security KA.

PBD-Platform-Based Development

PBD is a new KA that recognizes the increasing use of platform-specific programming environments, both at the introductory level and in upper-level electives. Platforms such as the Web or mobile devices enable students to learn within and about environments constrained by hardware, APIs, and special services (often in cross-disciplinary contexts). These environments are sufficiently different from “general purpose” programming to warrant this new (wholly elective) KA.

PD-Parallel and Distributed Computing

Previous curricular volumes had parallelism topics distributed across disparate KAs as electives. Given the vastly increased importance of parallel and distributed computing, it seemed crucial to identify essential concepts in this area and to promote those topics to the core. To highlight and coordinate this material, CS2013 dedicates a KA to this area. This new KA includes material on programming models, programming pragmatics, algorithms, performance, computer architecture, and distributed systems.

SDF-Software Development Fundamentals

This new KA generalizes introductory programming to focus on the entire software development process, identifying concepts and skills that should be mastered in the first year of a computer science program. As a result of its broad purpose, the SDF KA includes fundamental concepts and skills that could appear in other software-oriented KAs (e.g., programming constructs from Programming Languages, simple algorithm analysis from Algorithms and Complexity, simple development methodologies from Software Engineering). Likewise, each of those KAs will contain more advanced material that builds upon the fundamental concepts and skills in SDF. Compared to previous volumes, key approaches to programming -- including object-oriented programming, functional programming, and event-driven programming -- are kept in one place, namely the PL KA, even though many curricula will cover some of these topics in introductory courses.

SF-Systems Fundamentals

In previous curricular volumes, the interacting layers of a typical computing system, from hardware building blocks, to architectural organization, to operating system services, to application execution environments (particularly for parallel execution in a modern view of applications), were presented in independent knowledge units. The new Systems Fundamentals KA presents a unified systems perspective and common conceptual foundation for other KAs (notably Architecture and Organization, Network and Communications, Operating Systems, and Parallel and Distributed Algorithms). An organizational principle is “programming for performance”: what a programmer needs to understand about the underlying system to achieve high performance, particularly in terms of exploiting parallelism.

How to Read the Body of Knowledge

Curricular Hours

Continuing in the tradition of CC2001/CS2008, we define the unit of coverage in the Body of Knowledge in terms of **lecture hours**, as being the sole unit that is understandable in (and transferable to) cross-cultural contexts. An “hour” corresponds to the time required to present the material in a traditional lecture-oriented format; the hour count does not include any additional work that is associated with a lecture (e.g., in self-study, lab classes, assessments, etc.). Indeed, we expect students to spend a significant amount of additional time outside of class developing facility with the material presented in class. As with previous reports, we maintain the principle that the use of a lecture-hour as the unit of measurement does not require or endorse the use of traditional lectures for the presentation of material.

The specification of topic hours represents the **minimum** amount of time we expect such coverage to take. Any institution may opt to cover the same material in a longer period of time as warranted by the individual needs of that institution.

Courses

Throughout the Body of Knowledge, when we refer to a “course” we mean an institutionally-recognised unit of study. Depending on local circumstance, full-time students will take several “courses” at any one time, typically eight or more per academic year. While “course” is a common term at some institutions, others will use other names, for example “module” or “paper”.

Guidance on Learning Outcomes

Each KU within a KA lists both a set of topics and the learning outcomes students are expected to achieve with respect to the topics specified. Each learning outcome has a *level of mastery* associated with it. There are three levels of mastery, defined as:

- *Knowledge*: The student understands what a concept is or what it means. This level of mastery provides a basic awareness of a concept as opposed to expecting real facility with its application.
- *Application*: The student is able to apply a concept in a concrete way. Applying a concept may include, for example, the ability to implement a programming concept, use a particular proof technique, or perform a particular analysis.
- *Evaluation*: The student is able to consider a concept from multiple view points and/or justify the selection of a particular approach to solve a problem. This level of mastery implies more than the application of a concept; it involves the ability to select an appropriate approach from understood alternatives.

As a concrete, although admittedly simplistic, example of these levels of mastery, we consider the notion of iteration in software development, for example for-loops, while-loops, iterators. At the level of “Knowledge,” a student would be expected to know what the concept of iteration is in software development and why it is a useful technique. In order to show mastery at the “Application” level, a student should be able to write a program using a form of iteration. Understanding iteration at the “Evaluation” level would require a student to understand multiple methods for iteration and be able to appropriately select among them for different applications.

Core Hours in Knowledge Areas

An overview of the number of core hours (both Tier1 and Tier2) by KA in the CS2013 Body of Knowledge is provided below (for a discussion of Tier1 and Tier2, see Chapter 4). For comparison, the number of core hours from both the previous CS2008 and CC2001 reports are provided as well.

Knowledge Area	CS2013 Tier1	CS2013 Tier2	CS2008 Core	CC2001 Core
AL-Algorithms and Complexity	19	9	31	31
AR-Architecture and Organization	0	16	36	36
CN-Computational Science	1	0	0	0
DS-Discrete Structures	37	4	43	43
GV-Graphics and Visual Computing	2	1	3	3
HC-Human-Computer Interaction	4	4	8	8
IAS-Security and Information Assurance	2	6	--	--
IM-Information Management	1	9	11	10
IS-Intelligent Systems	0	10	10	10
NC-Networking and Communication	3	7	15	15
OS-Operating Systems	4	11	18	18
PBD-Platform-based Development	0	0	--	--
PD-Parallel and Distributed Computing	5	10	--	--
PL-Programming Languages	8	20	21	21
SDF-Software Development Fundamentals	42	0	47	38
SE-Software Engineering	6	21	31	31
SF-Systems Fundamentals	18	9	--	--
SP-Social and Professional Issues	11	5	16	16
Total Core Hours	**163**	**142**	**290**	**280**

All Tier1 + All Tier2 Total	**305**
All Tier1 + 90% of Tier2 Total	**290.8**
All Tier1 + 80% of Tier2 Total	**276.6**

As seen above, in CS2013 the total Tier1 hours together with the entirety of Tier2 hours slightly exceeds the total core hours from previous reports. However, it is important to note that the tiered structure of the core in CS2013 explicitly provides the flexibility for institutions to select topics from Tier2 (to include at least 80%). As a result, it is possible to implement the CS2013 guidelines with slightly fewer hours than previous curricular guidelines.

Appendix A: The Body of Knowledge

Algorithms and Complexity (AL)

Algorithms are fundamental to computer science and software engineering. The real-world performance of any software system depends on: (1) the algorithms chosen and (2) the suitability and efficiency of the various layers of implementation. Good algorithm design is therefore crucial for the performance of all software systems. Moreover, the study of algorithms provides insight into the intrinsic nature of the problem as well as possible solution techniques independent of programming language, programming paradigm, computer hardware, or any other implementation aspect.

An important part of computing is the ability to select algorithms appropriate to particular purposes and to apply them, recognizing the possibility that no suitable algorithm may exist. This facility relies on understanding the range of algorithms that address an important set of well-defined problems, recognizing their strengths and weaknesses, and their suitability in particular contexts. Efficiency is a pervasive theme throughout this area.

This knowledge area defines the central concepts and skills required to design, implement, and analyze algorithms for solving problems. Algorithms are essential in all advanced areas of computer science: artificial intelligence, databases, distributed computing, graphics, networking, operating systems, programming languages, security, and so on. Algorithms that have specific utility in each of these are listed in the relevant knowledge areas. Cryptography, for example, appears in the new knowledge area on Information Assurance and Security, while parallel and distributed algorithms appear in PD-Parallel and Distributed Computing.

As with all knowledge areas, the order of topics and their groupings do not necessarily correlate to a specific order of presentation. Different programs will teach the topics in different courses and should do so in the order they believe is most appropriate for their students.

AL. Algorithms and Complexity (19 Core-Tier1 hours, 9 Core-Tier2 hours)

	Core-Tier1 hours	Core-Tier2 hours	Includes Electives
AL/Basic Analysis	**2**	**2**	**N**
AL/Algorithmic Strategies	**5**	**1**	**N**
AL/Fundamental Data Structures and Algorithms	**9**	**3**	**N**
AL/Basic Automata, Computability and Complexity	**3**	**3**	**N**
AL/Advanced Computational Complexity			**Y**
AL/Advanced Automata Theory and Computability			**Y**
AL/Advanced Data Structures, Algorithms, and Analysis			**Y**

AL/Basic Analysis

[2 Core-Tier1 hours, 2 Core-Tier2 hours]

Topics:

[Core-Tier1]

- Differences among best, average, and worst case behaviors of an algorithm
- Asymptotic analysis of upper and average complexity bounds
- Big O notation: formal definition
- Complexity classes, such as constant, logarithmic, linear, quadratic, and exponential
- Empirical measurements of performance
- Time and space trade-offs in algorithms

[Core-Tier2]

- Big O notation: use
- Little o, big omega and big theta notation
- Recurrence relations and analysis of recursive algorithms
- Some version of a Master Theorem

Learning Outcomes:

1. Explain what is meant by “best”, “average”, and “worst” case behavior of an algorithm. [Knowledge]
2. In the context of specific algorithms, identify the characteristics of data and/or other conditions or assumptions that lead to different behaviors. [Evaluation]
3. Determine informally the time and space complexity of simple algorithms. [Application]
4. Understand the formal definition of big O. [Knowledge]
5. List and contrast standard complexity classes. [Knowledge]

6. Perform empirical studies to validate hypotheses about runtime stemming from mathematical analysis. Run algorithms on input of various sizes and compare performance. [Evaluation]
7. Give examples that illustrate time-space trade-offs of algorithms. [Knowledge]
8. Use big O notation formally to give asymptotic upper bounds on time and space complexity of algorithms. [Application]
9. Use big O notation formally to give average case bounds on time complexity of algorithms. [Application]
10. Explain the use of big omega, big theta, and little o notation to describe the amount of work done by an algorithm. [Knowledge]
11. Use recurrence relations to determine the time complexity of recursively defined algorithms. [Application]
12. Solve elementary recurrence relations, e.g., using some form of a Master Theorem. [Application]

AL/Algorithmic Strategies

[5 Core-Tier1 hours, 1 Core-Tier2 hours]

An instructor might choose to cover these algorithmic strategies in the context of the algorithms presented in "Fundamental Data Structures and Algorithms" below. While the total number of hours for the two knowledge units (18) could be divided differently between them, our sense is that the 1:2 ratio is reasonable.

Topics:

[Core-Tier1]

- Brute-force algorithms
- Greedy algorithms
- Divide-and-conquer (cross-reference SDF/Algorithms and Design/Problem-solving strategies)
- Recursive backtracking
- Dynamic Programming

[Core-Tier2]

- Branch-and-bound
- Heuristics
- Reduction: transform-and-conquer

Learning Outcomes:

1. For each of the above strategies, identify a practical example to which it would apply. [Knowledge]
2. Have facility mapping pseudocode to implementation, implementing examples of algorithmic strategies from scratch, and applying them to specific problems. [Application]
3. Use a greedy approach to solve an appropriate problem and determine if the greedy rule chosen leads to an optimal solution. [Application, Evaluation]
4. Use a divide-and-conquer algorithm to solve an appropriate problem. [Application]
5. Use recursive backtracking to solve a problem such as navigating a maze. [Application]
6. Use dynamic programming to solve an appropriate problem. [Application]
7. Describe various heuristic problem-solving methods. [Knowledge]
8. Use a heuristic approach to solve an appropriate problem. [Application]
9. Describe the trade-offs between brute force and other strategies. [Evaluation]

AL/Fundamental Data Structures and Algorithms

[9 Core-Tier1 hours, 3 Core-Tier2 hours]

This knowledge unit builds directly on the foundation provided by Software Development Fundamentals (SDF), particularly the material in SDF/Fundamental Data Structures and SDF/Algorithms and Design.

Topics:

[Core-Tier1]

Implementation and use of:

- Simple numerical algorithms, such as computing the average of a list of numbers, finding the min, max, and mode in a list, approximating the square root of a number, or finding the greatest common divisor
- Sequential and binary search algorithms
- Worst case quadratic sorting algorithms (selection, insertion)
- Worst or average case O(N log N) sorting algorithms (quicksort, heapsort, mergesort)
- Hash tables, including strategies for avoiding and resolving collisions
- Binary search trees
- Common operations on binary search trees such as select min, max, insert, delete, iterate over tree
- Graphs and graph algorithms
- Representations of graphs (e.g., adjacency list, adjacency matrix)
- Depth- and breadth-first traversals

[Core-Tier2]

- Graphs and graph algorithms
- Shortest-path algorithms (Dijkstra's and Floyd's algorithms)
- Minimum spanning tree (Prim's and Kruskal's algorithms)
- Pattern matching and string/text algorithms (e.g., substring matching, regular expression matching, longest common subsequence algorithms)

Learning Outcomes:

1. Implement basic numerical algorithms. [Application]
2. Implement simple search algorithms and explain the differences in their time complexities. [Application, Evaluation]
3. Be able to implement common quadratic and O(N log N) sorting algorithms. [Application]
4. Understand the implementation of hash tables, including collision avoidance and resolution. [Knowledge]
5. Discuss the runtime and memory efficiency of principal algorithms for sorting, searching, and hashing. [Knowledge]
6. Discuss factors other than computational efficiency that influence the choice of algorithms, such as programming time, maintainability, and the use of application-specific patterns in the input data. [Knowledge]
7. Solve problems using fundamental graph algorithms, including depth-first and breadth-first search. [Application]
8. Demonstrate the ability to evaluate algorithms, to select from a range of possible options, to provide justification for that selection, and to implement the algorithm in a particular context. [Application, Evaluation]
9. Solve problems using graph algorithms, including single-source and all-pairs shortest paths, and at least one minimum spanning tree algorithm. [Application]
10. Be able to implement a string-matching algorithm. [Application]

AL/Basic Automata Computability and Complexity

[3 Core-Tier1 hours, 3 Core-Tier2 hours]

Topics:

[Core-Tier1]

- Finite-state machines
- Regular expressions
- The halting problem

[Core-Tier2]

- Context-free grammars (cross-reference PL/Syntax Analysis)
- P vs NP (tractable and intractable problems)
- Definition of P, NP, and NP-complete
- Exemplary NP-complete problems (e.g., SAT, Knapsack)

Learning Outcomes:

1. Discuss the concept of finite state machines. [Knowledge]
2. Design a deterministic finite state machine to accept a specified language. [Application]
3. Generate a regular expression to represent a specified language. [Application]
4. Explain why the halting problem has no algorithmic solution. [Knowledge]
5. Design a context-free grammar to represent a specified language. [Application]
6. Define the classes P and NP. [Knowledge]
7. Explain the significance of NP-completeness. [Knowledge]

AL/Advanced Computational Complexity

[Elective]

Topics:

- Review definitions of the classes P and NP; introduce EXP
- NP-completeness (Cook's theorem)
- Classic NP-complete problems
- Reduction Techniques

Learning Outcomes:

1. Define the classes P and NP. (Also appears in AL/Basic Automata, Computability, and Complexity) [Knowledge]
2. Define the class EXP. [Knowledge]
3. Explain the significance of NP-completeness. (Also appears in AL/Basic Automata, Computability, and Complexity) [Knowledge]
4. Provide examples of classic NP-complete problems. [Knowledge]
5. Prove that a problem is NP-complete by reducing a classic known NP-complete problem to it. [Application]

AL/Advanced Automata Theory and Computability

[Elective]

Topics:

- Sets and languages
- Regular languages
- Review of deterministic finite automata (DFAs)
- Nondeterministic finite automata (NFAs)
- Equivalence of DFAs and NFAs
- Review of regular expressions; their equivalence to finite automata
- Closure properties
- Proving languages non-regular, via the pumping lemma or alternative means
- Context-free languages
- Push-down automata (PDAs)
- Relationship of PDAs and context-free grammars
- Properties of context-free languages
- Turing machines, or an equivalent formal model of universal computation
- Nondeterministic Turing machines
- Chomsky hierarchy
- The Church-Turing thesis
- Computability
- Rice's Theorem
- Examples of uncomputable functions
- Implications of uncomputability

Learning Outcomes:

1. Determine a language's place in the Chomsky hierarchy (regular, context-free, recursively enumerable). [Evaluation]
2. Prove that a language is in a specified class and that it is not in the next lower class. [Evaluation]
3. Convert among equivalently powerful notations for a language, including among DFAs, NFAs, and regular expressions, and between PDAs and CFGs. [Application]
4. Explain the Church-Turing thesis and its significance. [Knowledge]
5. Explain Rice's Theorem and its significance. [Knowledge]
6. Provide examples of uncomputable functions. [Knowledge]
7. Prove that a problem is uncomputable by reducing a classic known uncomputable problem to it. [Application]

AL/Advanced Data Structures Algorithms and Analysis

[Elective]

Many programs will want their students to have exposure to more advanced algorithms or methods of analysis. Below is a selection of possible advanced topics that are current and timely but by no means exhaustive.

Topics:

- Balanced trees (e.g., AVL trees, red-black trees, splay trees, treaps)
- Graphs (e.g., topological sort, Tarjan's algorithm, matching)
- Advanced data structures (e.g., B-trees, tries, Fibonacci heaps)

- Network flows (e.g., max flow [Ford-Fulkerson algorithm], max flow – min cut, maximum bipartite matching)
- Linear Programming (e.g., duality, simplex method, interior point algorithms)
- Number-theoretic algorithms (e.g., modular arithmetic, primality testing, integer factorization)
- Geometric algorithms (e.g., points, line segments, polygons [properties, intersections], finding convex hull, spatial decomposition, collision detection, geometric search/proximity)
- Randomized algorithms
- Approximation algorithms
- Amortized analysis
- Probabilistic analysis
- Online algorithms and competitive analysis

Learning Outcomes:

1. Understand the mapping of real-world problems to algorithmic solutions (e.g., as graph problems, linear programs, etc.) [Application, Evaluation]
2. Use advanced algorithmic techniques (e.g., randomization, approximation) to solve real problems. [Application]
3. Apply advanced analysis techniques (e.g., amortized, probabilistic, etc.) to algorithms.

Architecture and Organization (AR)

Computing professionals should not regard the computer as just a black box that executes programs by magic. AR builds on SF to develop a deeper understanding of the hardware environment upon which all of computing is based, and the interface it provides to higher software layers. Students should acquire an understanding and appreciation of a computer system's functional components, their characteristics, performance, and interactions, and, in particular, the challenge of harnessing parallelism to sustain performance improvements now and into the future. Students need to understand computer architecture to develop programs that can achieve high performance through a programmer's awareness of parallelism and latency. In selecting a system to use, students should to able to understand the tradeoff among various components, such as CPU clock speed, cycles per instruction, memory size, and average memory access time.

The learning outcomes specified for these topics correspond primarily to the core and are intended to support programs that elect to require only the minimum 16 hours of computer architecture of their students. For programs that want to teach more than the minimum, the same topics (AR1-AR8) can be treated at a more advanced level by implementing a two-course sequence. For programs that want to cover the elective topics, those topics can be introduced within a two-course sequence and/or be treated in a more comprehensive way in a third course.

AR. Architecture and Organization (0 Core-Tier 1 hours, 16 Core-Tier 2 hours)

	Core-Tier 1 hours	Core-Tier 2 Hours	Includes Elective
AR/Digital logic and digital systems		3	N
AR/Machine level representation of data		3	N
AR/Assembly level machine organization		6	N
AR/Memory system organization and architecture		3	N
AR/Interfacing and communication		1	N
AR/Functional organization			Y
AR/Multiprocessing and alternative architectures			Y
AR/Performance enhancements			Y

AR/Digital logic and digital systems

[3 Core-Tier 2 hours]

Topics:

- Overview and history of computer architecture
- Combinational vs. sequential logic/Field programmable gate arrays as a fundamental combinational + sequential logic building block
- Multiple representations/layers of interpretation (hardware is just another layer)
- Computer-aided design tools that process hardware and architectural representations
- Register transfer notation/Hardware Description Language (Verilog/VHDL)
- Physical constraints (gate delays, fan-in, fan-out, energy/power)

Learning outcomes:

1. Describe the progression of computer technology components from vacuum tubes to VLSI, from mainframe computer architectures to the organization of warehouse-scale computers [Knowledge].
2. Comprehend the trend of modern computer architectures towards multi-core and that parallelism is inherent in all hardware systems [Knowledge].
3. Explain the implications of the "power wall" in terms of further processor performance improvements and the drive towards harnessing parallelism [Knowledge].
4. Articulate that there are many equivalent representations of computer functionality, including logical expressions and gates, and be able to use mathematical expressions to describe the functions of simple combinational and sequential circuits [Knowledge].
5. Design the basic building blocks of a computer: arithmetic-logic unit (gate-level), registers (gate-level), central processing unit (register transfer-level), memory (register transfer-level) [Application].
6. Use CAD tools for capture, synthesis, and simulation to evaluate simple building blocks (e.g., arithmetic-logic unit, registers, movement between registers) of a simple computer design [Application].

7. Evaluate the functional and timing diagram behavior of a simple processor implemented at the logic circuit level [Evaluation].

AR/Machine-level representation of data

[3 Core-Tier 2 hours]

Topics:

- Bits, bytes, and words
- Numeric data representation and number bases
- Fixed- and floating-point systems
- Signed and twos-complement representations
- Representation of non-numeric data (character codes, graphical data)
- Representation of records and arrays

Learning outcomes:

1. Explain why everything is data, including instructions, in computers [Knowledge].
2. Explain the reasons for using alternative formats to represent numerical data [Knowledge].
3. Describe how negative integers are stored in sign-magnitude and twos-complement representations [Knowledge].
4. Explain how fixed-length number representations affect accuracy and precision [Knowledge].
5. Describe the internal representation of non-numeric data, such as characters, strings, records, and arrays [Knowledge].
6. Convert numerical data from one format to another [Application].
7. Write simple programs at the assembly/machine level for string processing and manipulation [Application].

AR/Assembly level machine organization

[6 Core-Tier 2 hours]

Topics:

- Basic organization of the von Neumann machine
- Control unit; instruction fetch, decode, and execution
- Instruction sets and types (data manipulation, control, I/O)
- Assembly/machine language programming
- Instruction formats
- Addressing modes
- Subroutine call and return mechanisms
- I/O and interrupts
- Heap vs. Static vs. Stack vs. Code segments
- Shared memory multiprocessors/multicore organization
- Introduction to SIMD vs. MIMD and the Flynn Taxonomy

***Learning outcomes*:**

1. Explain the organization of the classical von Neumann machine and its major functional units [Knowledge].
2. Describe how an instruction is executed in a classical von Neumann machine, with extensions for threads, multiprocessor synchronization, and SIMD execution [Knowledge].

3. Describe instruction level parallelism and hazards, and how they are managed in typical processor pipelines [Knowledge].
4. Summarize how instructions are represented at both the machine level and in the context of a symbolic assembler [Knowledge].
5. Demonstrate how to map between high-level language patterns into assembly/machine language notations [Knowledge].
6. Explain different instruction formats, such as addresses per instruction and variable length vs. fixed length formats [Knowledge].
7. Explain how subroutine calls are handled at the assembly level [Knowledge].
8. Explain the basic concepts of interrupts and I/O operations [Knowledge].
9. Explain how subroutine calls are handled at the assembly level [Knowledge].
10. Write simple assembly language program segments [Application].
11. Show how fundamental high-level programming constructs are implemented at the machine-language level [Application].

AR/Memory system organization and architecture

[3 Core-Tier 2 hours]

[Cross-reference OS/Memory Management--Virtual Machines]

Topics:

- Storage systems and their technology
- Memory hierarchy: importance of temporal and spatial locality
- Main memory organization and operations
- Latency, cycle time, bandwidth, and interleaving
- Cache memories (address mapping, block size, replacement and store policy)
- Multiprocessor cache consistency/Using the memory system for inter-core synchronization/atomic memory operations
- Virtual memory (page table, TLB)
- Fault handling and reliability
- Coding, data compression, and data integrity

Learning outcomes:

1. Identify the main types of memory technology [Knowledge].
2. Explain the effect of memory latency on running time [Knowledge].
3. Describe how the use of memory hierarchy (cache, virtual memory) is used to reduce the effective memory latency [Knowledge].
4. Describe the principles of memory management [Knowledge].
5. Explain the workings of a system with virtual memory management [Knowledge].
6. Compute Average Memory Access Time under a variety of memory system configurations and workload assumptions [Application].

AR/Interfacing and communication

[1 Core-Tier 2 hour]

[Cross-reference OS Knowledge Area for a discussion of the operating system view of input/output processing and management. The focus here is on the hardware mechanisms for supporting device interfacing and processor-to-processor communications.]

Topics:

- I/O fundamentals: handshaking, buffering, programmed I/O, interrupt-driven I/O
- Interrupt structures: vectored and prioritized, interrupt acknowledgment
- External storage, physical organization, and drives
- Buses: bus protocols, arbitration, direct-memory access (DMA)
- Introduction to networks: networks as another layer of access hierarchy
- Multimedia support
- RAID architectures

***Learning outcomes*:**

1. Explain how interrupts are used to implement I/O control and data transfers [Knowledge].
2. Identify various types of buses in a computer system [Knowledge].
3. Describe data access from a magnetic disk drive [Knowledge].
4. Compare common network organizations, such as ethernet/bus, ring, switched vs. routed [Knowledge].
5. Identify interfaces needed for multimedia support, from storage, through network, to memory and display [Knowledge].
6. Describe the advantages and limitations of RAID architectures [Knowledge].

AR/Functional organization

[Elective]

[Note: elective for computer scientist; would be core for computer engineering curriculum]

Topics:

- Implementation of simple datapaths, including instruction pipelining, hazard detection and resolution
- Control unit: hardwired realization vs. microprogrammed realization
- Instruction pipelining
- Introduction to instruction-level parallelism (ILP)

Learning outcomes:

1. Compare alternative implementation of datapaths [Knowledge].
2. Discuss the concept of control points and the generation of control signals using hardwired or microprogrammed implementations [Knowledge].
3. Explain basic instruction level parallelism using pipelining and the major hazards that may occur [Knowledge].
4. Design and implement a complete processor, including datapath and control [Application].
5. Determine, for a given processor and memory system implementation, the average cycles per instruction [Evaluation].

AR/Multiprocessing and alternative architectures

[Elective]

[Cross-reference PD/Parallel Architecture: The view here is on the hardware implementation of SIMD and MIMD architectures; in PD/Parallel Architecture, it is on the way that algorithms can be matched to the underlying hardware capabilities for these kinds of parallel processing architectures.]

Topics:

- Power Law: Energy as a limiting factor in processor design
- Example SIMD and MIMD instruction sets and architectures
- Interconnection networks (hypercube, shuffle-exchange, mesh, crossbar)
- Shared multiprocessor memory systems and memory consistency
- Multiprocessor cache coherence

***Learning outcomes*:**

1. Discuss the concept of parallel processing beyond the classical von Neumann model [Knowledge].
2. Describe alternative architectures such as SIMD and MIMD [Knowledge].
3. Explain the concept of interconnection networks and characterize different approaches [Knowledge].
4. Discuss the special concerns that multiprocessing systems present with respect to memory management and describe how these are addressed [Knowledge].
5. Describe the differences between memory backplane, processor memory interconnect, and remote memory via networks [Knowledge].

AR/Performance enhancements

[Elective]

Topics:

- Superscalar architecture
- Branch prediction, Speculative execution, Out-of-order execution
- Prefetching
- Vector processors and GPUs
- Hardware support for Multithreading
- Scalability
- Alternative architectures, such as VLIW/EPIC, and Accelerators and other kinds of Special-Purpose Processors

Learning outcomes:

1. Describe superscalar architectures and their advantages [Knowledge].
2. Explain the concept of branch prediction and its utility [Knowledge].
3. Characterize the costs and benefits of prefetching [Knowledge].
4. Explain speculative execution and identify the conditions that justify it [Knowledge].
5. Discuss the performance advantages that multithreading offered in an architecture along with the factors that make it difficult to derive maximum benefits from this approach [Knowledge].
6. Describe the relevance of scalability to performance [Knowledge].

Computational Science (CN)

Computational Science is a field of applied computer science, that is, the application of computer science to solve problems across a range of disciplines. According to the book “Introduction to Computational Science”, Shiflet & Shiflet offer the following definition: “the field of computational science combines computer simulation, scientific visualization, mathematical modeling, computer programming and data structures, networking, database design, symbolic computation, and high performance computing with various disciplines.”Computer science, which largely focuses on the theory, design, and implementation of algorithms for manipulating data and information, can trace its roots to the earliest devices used to assist people in computation over four thousand years ago. Various systems were created and used to calculate astronomical positions. Ada Lovelace’s programming achievement was intended to calculate Bernoulli numbers. In the late nineteenth century, mechanical calculators became available, and were immediately put to use by scientists. The needs of scientists and engineers for computation have long driven research and innovation in computing. As computers increase in their problem-solving power, computational science has grown in both breadth and importance. It is a discipline in its own right (President’s Information Technology Advisory Committee, 2005, page 13) and is considered to be “one of the five college majors on the rise” (Fischer and Gleen, “5 College Majors on the Rise”, The Chronicle of Higher Education, 2009.) An amazing assortment of sub-fields have arisen under the umbrella of Computational Science, including computational biology, computational chemistry, computational mechanics, computational archeology, computational finance, computational sociology and computational forensics.

Some fundamental concepts of computational science are germane to every computer scientist, and computational science topics are extremely valuable components of an undergraduate program in computer science. This area offers exposure to many valuable ideas and techniques, including precision of numerical representation, error analysis, numerical techniques, parallel architectures and algorithms, modeling and simulation, information visualization, software engineering, and optimization. At the same time, students who take courses in this area have an opportunity to apply these techniques in a wide range of application areas, such as: molecular and fluid dynamics, celestial mechanics, economics, biology, geology, medicine, and social network analysis.

In the computational science community, the terms *run*, *modify*, and *create* are often used to describe levels of understanding. This chapter follows the conventions of other chapters in this volume and uses the terms *knowledge*, *application*, and *evaluation*.

CN. Computational Science (1 Core-Tier1 hours, 0 Core-Tier2 hours)

	Core-Tier1 hours	Core-Tier2 hours	Includes Electives
CN/Fundamentals	***1***		**N**
CN/Modeling and Simulation			**Y**
CN/Processing			**Y**
CN/Interactive Visualization			**Y**
CN/Data, Information, and Knowledge			**Y**

CN/Fundamentals

[1 Core-Tier1 hours]

This describes part of the abstraction that computer scientists do. The real world doesn't fit in the machine, so we have to abstract, simulate, and model the world in order to make the machine do something useful. This is a principal approach to computing. This can be thought of as where the field came from: modeling things such as trajectories of artillery shells, which was the impetus for building the Eniac at the Moore School of the University of Pennsylvania in the mid-1940's.

Modeling and simulation are essential topics for computational science. Any introduction to computational science would either include or presume an introduction to computing. Topics relevant to computational science include fundamental concepts in program construction (SDF/Fundamental Programming Concepts), algorithm design (SDF/Algorithms and Design), program testing (SDF/Development Methods), data representations (AR/Machine Representation of Data), and basic computer architecture (AR/Memory System Organization and Architecture). In addition, a general set of modeling and simulation techniques, data visualization methods, and software testing and evaluation mechanisms are also important CN fundamentals.

Topics:

- Introduction to modeling and simulation
- Simulation techniques and tools, such as physical simulations, human-in-the-loop guided simulations, and virtual reality.
- Foundational approaches to validating models

Learning Outcomes:

1. Explain the concept of modeling and the use of abstraction that allows the use of a machine to solve a problem. [knowledge]
2. Explain the concept of simulation. [knowledge]
3. Describe the relationship between modeling and simulation, i.e., thinking of simulation as dynamic modeling. [knowledge]
4. Articulate the use of a formal mathematical model of a situation in the validation of a simulation. [knowledge]
5. Differentiate among the different types of simulations. [knowledge]
6. Describe several approaches to validating models. [knowledge]

CN/Modeling and Simulation

[Elective]

Topics:

- Purpose of modeling and simulation including optimization; supporting decision making, forecasting, safety considerations; for training and education.
- Tradeoffs including performance, accuracy, validity, and complexity.
- The simulation process; identification of key characteristics or behaviors, simplifying assumptions; validation of outcomes.
- Model building: use of mathematical formula or equation, graphs, constraints; methodologies and techniques; use of time stepping for dynamic systems.

- Formal models and modeling techniques: mathematical descriptions involving simplifying assumptions and avoiding detail. The descriptions use fundamental mathematical concepts such as set and function. Random numbers. Examples of techniques including:
- Monte Carlo methods
- Stochastic processes
- Queuing theory
- Petri nets and colored Petri nets
- Graph structures such as directed graphs, trees, networks
- Games, game theory, the modeling of things using game theory
- Linear programming and its extensions
- Dynamic programming
- Differential equations: ODE, PDE
- Non-linear techniques
- State spaces and transitions
- Assessing and evaluating models and simulations in a variety of contexts; verification and validation of models and simulations.
- Important application areas including health care and diagnostics, economics and finance, city and urban planning, science, and engineering.
- Software in support of simulation and modeling; packages, languages.

Learning Outcomes:

1. Explain and give examples of the benefits of simulation and modeling in a range of important application areas.
2. Demonstrate the ability to apply the techniques of modeling and simulation to a range of problem areas.
3. Explain the constructs and concepts of a particular modeling approach.
4. Explain the difference between validation and verification of a model; demonstrate the difference with specific examples[1].
5. Verify and validate the results of a simulation.
6. Evaluate a simulation, highlighting the benefits and the drawbacks.
7. Choose an appropriate modeling approach for a given problem or situation.
8. Compare results from different simulations of the same situation and explain any differences.
9. Infer the behavior of a system from the results of a simulation of the system.
10. Extend or adapt an existing model to a new situation.

[1] *Verification* means that the computations of the model are correct. If we claim to compute total time, for example, the computation actually does that. *Validation* asks whether the model matches the real situation.

CN/Processing

[Elective]

The processing topic area includes numerous topics from other knowledge areas. Specifically, coverage of processing should include a discussion of hardware architectures, including parallel systems, memory hierarchies, and interconnections among processors. These are covered in AR/Interfacing and Communication, AR/Multiprocessing and Alternative Architectures, AR/Performance Enhancements.

Topics:

- Fundamental programming concepts:
- The concept of an algorithm consisting of a finite number of well-defined steps, each of which completes in a finite amount of time, as does the entire process.
- Examples of well-known algorithms such as sorting and searching.
- The concept of analysis as understanding what the problem is really asking, how a problem can be approached using an algorithm, and how information is represented so that a machine can process it.
- The development or identification of a workflow.
- The process of converting an algorithm to machine-executable code.
- Software processes including lifecycle models, requirements, design, implementation, verification and maintenance.
- Machine representation of data computer arithmetic, and numerical methods, specifically sequential and parallel architectures and computations.
- Fundamental properties of parallel and distributed computation:
- Bandwidth.
- Latency.
- Scalability.
- Granularity.
- Parallelism including task, data, and event parallelism.
- Parallel architectures including processor architectures, memory and caching.
- Parallel programming paradigms including threading, message passing, event driven techniques, parallel software architectures, and MapReduce.
- Grid computing.
- The impact of architecture on computational time.
- Total time to science curve for parallelism: continuum of things.
- Computing costs, e.g., the cost of re-computing a value vs. the cost of storing and lookup.

Learning Outcomes:

1. Explain the characteristics and defining properties of algorithms and how they relate to machine processing.
2. Analyze simple problem statements to identify relevant information and select appropriate processing to solve the problem.
3. Identify or sketch a workflow for an existing computational process such as the creation of a graph based on experimental data.
4. Describe the process of converting an algorithm to machine-executable code.
5. Summarize the phases of software development and compare several common lifecycle models.
6. Explain how data is represented in a machine. Compare representations of integers to floating point numbers. Describe underflow, overflow, round off, and truncation errors in data representations.
7. Apply standard numerical algorithms to solve ODEs and PDEs. Use computing systems to solve systems of equations.
8. Describe the basic properties of bandwidth, latency, scalability and granularity.
9. Describe the levels of parallelism including task, data, and event parallelism.

10. Compare and contrast parallel programming paradigms recognizing the strengths and weaknesses of each.
11. Identify the issues impacting correctness and efficiency of a computation.
12. Design, code, test and debug programs for a parallel computation.

CN/Interactive Visualization

[Elective]

This sub-area is related to modeling and simulation. Most topics are discussed in detail in other knowledge areas in this document. There are many ways to present data and information, including immersion, realism, variable perspectives; haptics and heads-up displays, sonification, and gesture mapping.

Interactive visualization in general requires understanding of human perception (GV/Basics); graphics pipelines, geometric representations and data structures (GV/Fundamental Concepts); 2D and 3D rendering, surface and volume rendering (GV/Rendering, GV/Modeling, and GV/Advanced Rendering); and the use of APIs for developing user interfaces using standard input components such as menus, sliders, and buttons; and standard output components for data display, including charts, graphs, tables, and histograms (HCI/GUI Construction, HCI/GUI Programming).

Topics:

- Principles of data visualization.
- Graphing and visualization algorithms.
- Image processing techniques.
- Scalability concerns.

Learning Outcomes:

1. Compare common computer interface mechanisms with respect to ease-of-use, learnability, and cost.
2. Use standard APIs and tools to create visual displays of data, including graphs, charts, tables, and histograms.
3. Describe several approaches to using a computer as a means for interacting with and processing data.
4. Extract useful information from a dataset.
5. Analyze and select visualization techniques for specific problems.
6. Describe issues related to scaling data analysis from small to large data sets.

CN/Data, Information, and Knowledge

[Elective]

Many topics are discussed in detail in other knowledge areas in this document, specifically Information Management (IM/Information Management Concepts, IM/Database Systems, and IM/Data Modeling), Algorithms and Complexity (AL/Basic Analysis, AL/Fundamental Data Structures and Algorithms), and Software Development Fundamentals (SDF/Fundamental Programming Concepts, SDF/Development Methods).

Topics:

- Content management models, frameworks, systems, design methods (as in IM. Information Management).
- Digital representations of content including numbers, text, images (e.g., raster and vector), video (e.g., QuickTime, MPEG2, MPEG4), audio (e.g., written score, MIDI, sampled digitized sound track) and animations; complex/composite/aggregate objects; FRBR.
- Digital content creation/capture and preservation, including digitization, sampling, compression, conversion, transformation/translation, migration/emulation, crawling, harvesting.
- Content structure / management, including digital libraries and static/dynamic/stream aspects for:
- Data: data structures, databases.
- Information: document collections, multimedia pools, hyperbases (hypertext, hypermedia), catalogs, repositories.
- Knowledge: ontologies, triple stores, semantic networks, rules.
- Processing and pattern recognition, including indexing, searching (including: queries and query languages; central / federated / P2P), retrieving, clustering, classifying/categorizing, analyzing/mining/extracting, rendering, reporting, handling transactions.
- User / society support for presentation and interaction, including browse, search, filter, route, visualize, share, collaborate, rate, annotate, personalize, recommend.
- Modeling, design, logical and physical implementation, using relevant systems/software.

Learning Outcomes:

1. Identify all of the data, information, and knowledge elements and related organizations, for a computational science application.
2. Describe how to represent data and information for processing.
3. Describe typical user requirements regarding that data, information, and knowledge.
4. Select a suitable system or software implementation to manage data, information, and knowledge.
5. List and describe the reports, transactions, and other processing needed for a computational science application.
6. Compare and contrast database management, information retrieval, and digital library systems with regard to handling typical computational science applications.
7. Design a digital library for some computational science users / societies, with appropriate content and services.

Discrete Structures (DS)

Discrete structures are foundational material for computer science. By foundational we mean that relatively few computer scientists will be working primarily on discrete structures, but that many other areas of computer science require the ability to work with concepts from discrete structures. Discrete structures include important material from such areas as set theory, logic, graph theory, and probability theory.

The material in discrete structures is pervasive in the areas of data structures and algorithms but appears elsewhere in computer science as well. For example, an ability to create and understand a proof—either a formal symbolic proof or a less formal but still mathematically rigorous argument—is important in virtually every area of computer science, including (to name just a few) formal specification, verification, databases, and cryptography. Graph theory concepts are used in networks, operating systems, and compilers. Set theory concepts are used in software engineering and in databases. Probability theory is used in intelligent systems, networking, and a number of computing applications.

Given that discrete structures serves as a foundation for many other areas in computing, it is worth noting that the boundary between discrete structures and other areas, particularly Algorithms and Complexity, Software Development Fundamentals, Programming Languages, and Intelligent Systems, may not always be crisp. Indeed, different institutions may choose to organize the courses in which they cover this material in very different ways. Some institutions may cover these topics in one or two focused courses with titles like "discrete structures" or "discrete mathematics", whereas others may integrate these topics in courses on programming, algorithms, and/or artificial intelligence. Combinations of these approaches are also prevalent (e.g., covering many of these topics in a single focused introductory course and covering the remaining topics in more advanced topical courses).

DS. Discrete Structures (37 Core-Tier1 hours, 4 Core-Tier2 hours)

	Core-Tier1 hours	Core-Tier2 hours	Includes Electives
DS/Sets, Relations, and Functions	4		N
DS/Basic Logic	9		N
DS/Proof Techniques	10	1	N
DS/Basics of Counting	5		N
DS/Graphs and Trees	3	1	N
DS/Discrete Probability	6	2	N

DS/Sets, Relations, and Functions

[4 Core-Tier1 hours]

Topics:

[Core-Tier1]

- Sets
- Venn diagrams
- Union, intersection, complement
- Cartesian product
- Power sets
- Cardinality of finite sets
- Relations
- Reflexivity, symmetry, transitivity
- Equivalence relations, partial orders
- Functions
- Surjections, injections, bijections
- Inverses
- Composition

Learning Outcomes:

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

DS/Basic Logic

[9 Core-Tier1 hours]

Topics:

[Core-Tier1]

- Propositional logic (cross-reference: Propositional logic is also reviewed in IS/Knowledge Based Reasoning)
- Logical connectives
- Truth tables
- Normal forms (conjunctive and disjunctive)
- Validity
- Propositional inference rules (concepts of modus ponens and modus tollens)
- Predicate logic
- Universal and existential quantification
- Limitations of propositional and predicate logic (e.g., expressiveness issues)

Learning Outcomes:

1. Convert logical statements from informal language to propositional and predicate logic expressions. [Application]
2. Apply formal methods of symbolic propositional and predicate logic, such as calculating validity of formulae and computing normal forms. [Application]
3. Use the rules of inference to construct proofs in propositional and predicate logic. [Application]
4. Describe how symbolic logic can be used to model real-life situations or applications, including those arising in computing contexts such as software analysis (e.g., program correctness), database queries, and algorithms. [Application]
5. Apply formal logic proofs and/or informal, but rigorous, logical reasoning to real problems, such as predicting the behavior of software or solving problems such as puzzles. [Application]
6. Describe the strengths and limitations of propositional and predicate logic. [Knowledge]

DS/Proof Techniques

[10 Core-Tier1 hours, 1 Core-Tier2 hour]

Topics:

[Core-Tier1]

- Notions of implication, equivalence, converse, inverse, contrapositive, negation, and contradiction
- The structure of mathematical proofs
- Direct proofs
- Disproving by counterexample
- Proof by contradiction
- Induction over natural numbers
- Structural induction
- Weak and strong induction (i.e., First and Second Principle of Induction)
- Recursive mathematical definitions

[Core-Tier2]

- Well orderings

Learning Outcomes:

1. Identify the proof technique used in a given proof. [Knowledge]
2. Outline the basic structure of each proof technique described in this unit. [Application]
3. Apply each of the proof techniques correctly in the construction of a sound argument. [Application]
4. Determine which type of proof is best for a given problem. [Evaluation]
5. Explain the parallels between ideas of mathematical and/or structural induction to recursion and recursively defined structures. [Evaluation]
6. Explain the relationship between weak and strong induction and give examples of the appropriate use of each. [Evaluation]

DS/Basics of Counting

[5 Core-Tier1 hours]

Topics:

[Core-Tier1]

- Counting arguments
- Set cardinality and counting
- Sum and product rule
- Inclusion-exclusion principle
- Arithmetic and geometric progressions
- The pigeonhole principle
- Permutations and combinations
- Basic definitions
- Pascal’s identity
- The binomial theorem
- Solving recurrence relations (cross-reference: AL/Basic Analysis)
- An example of a simple recurrence relation, such as Fibonacci numbers
- Other examples, showing a variety of solutions
- Basic modular arithmetic

Learning Outcomes:

1. Apply counting arguments, including sum and product rules, inclusion-exclusion principle and arithmetic/geometric progressions. [Application]
2. Apply the pigeonhole principle in the context of a formal proof. [Application]
3. Compute permutations and combinations of a set, and interpret the meaning in the context of the particular application. [Application]
4. Map real-world applications to appropriate counting formalisms, such as determining the number of ways to arrange people around a table, subject to constraints on the seating arrangement, or the number of ways to determine certain hands in cards (e.g., a full house). [Application]
5. Solve a variety of basic recurrence relations. [Application]
6. Analyze a problem to determine underlying recurrence relations. [Application]
7. Perform computations involving modular arithmetic. [Application]

DS/Graphs and Trees

[3 Core-Tier1 hours, 1 Core-Tier2 hour]

(cross-reference: AL/Fundamental Data Structures and Algorithms)

Topics:

[Core-Tier1]

- Trees
- Undirected graphs
- Directed graphs
- Weighted graphs
- Traversal strategies

[Core-Tier2]

- Spanning trees/forests
- Graph isomorphism

Learning Outcomes:

1. Illustrate by example the basic terminology of graph theory, and some of the properties and special cases of each type of graph/tree. [Knowledge]
2. Demonstrate different traversal methods for trees and graphs, including pre, post, and in-order traversal of trees. [Application]
3. Model *a variety of* real-world problems in computer science using appropriate forms of graphs and trees, such as representing a network topology or the organization of a hierarchical file system. [Application]
4. Show how concepts from graphs and trees appear in data structures, algorithms, proof techniques (structural induction), and counting. [Application]

DS/Discrete Probability

[6 Core-Tier1 hours, 2 Core-Tier2 hour]

(Cross-reference IS/Basic Knowledge Representation and Reasoning, which includes a review of basic probability)

Topics:

[Core-Tier1]

- Finite probability space, events
- Axioms of probability and probability measures
- Conditional probability, Bayes' theorem
- Independence
- Integer random variables (Bernoulli, binomial)
- Expectation, including Linearity of Expectation

[Core-Tier2]

- Variance
- Conditional Independence

Learning Outcomes:

1. Calculate probabilities of events and expectations of random variables for elementary problems such as games of chance. [Application]
2. Differentiate between dependent and independent events. [Application]
3. Explain how events that are independent can be conditionally dependent (and vice-versa). Identify real-world examples of such cases. [Application]
4. Identify a case of the binomial distribution and compute a probability using that distribution. [Application]
5. Make a probabilistic inference in a real-world problem using Bayes' theorem to determine the probability of a hypothesis given evidence. [Application]
6. Apply the tools of probability to solve problems such as the average case analysis of algorithms or analyzing hashing. [Application]

Graphics and Visualization (GV)

Computer graphics is the term commonly used to describe the computer generation and manipulation of images. It is the science of enabling visual communication through computation. Its uses include cartoons, film special effects, video games, medical imaging, engineering, as well as scientific,   information, and knowledge visualization. Traditionally, graphics at the undergraduate level has focused on rendering, linear algebra, and phenomenological approaches. More recently, the focus has begun to include physics, numerical integration, scalability, and special-purpose hardware, In order for students to become adept at the use and generation of computer graphics, many implementation-specific issues must be addressed, such as file formats, hardware interfaces, and application program interfaces. These issues change rapidly, and the description that follows attempts to avoid being overly prescriptive about them. The area encompassed by Graphics and Visual Computing (GV) is divided into several interrelated fields:

- Fundamentals: Computer graphics depends on an understanding of how humans use vision to perceive information and how information can be rendered on a display device. Every computer scientist should have some understanding of where and how graphics can be appropriately applied and the fundamental processes involved in display rendering.
- Modeling: Information to be displayed must be encoded in computer memory in some form, often in the form of a mathematical specification of shape and form.
- Rendering: Rendering is the process of displaying the information contained in a model.
- Animation: Animation is the rendering in a manner that makes images appear to move and the synthesis or acquisition of the time variations of models.
- Visualization. The field of visualization seeks to determine and present underlying correlated structures and relationships in data sets from a wide variety of application areas. The prime objective of the presentation should be to communicate the information in a dataset so as to enhance understanding
- Computational Geometry: Computational Geometry is the study of algorithms that are stated in terms of geometry.

Graphics and Visualization is related to machine vision and image processing (in the Intelligent Systems KA) and algorithms such as computational geometry, which can be found in the Algorithms and Complexity KA. Topics in virtual reality can be found in the Human Computer Interaction KA.

This description assumes students are familiar with fundamental concepts of data representation, abstraction, and program implementation.

GV. Graphics and Visualization (2 Core-Tier1 hours, 1 Core-Tier2 hours)

	Core-Tier1 hours	Core-Tier2 hours	Includes Electives
GV/Fundamental Concepts	2	1	N
GV/Basic Rendering			Y
GV/Geometric Modeling			Y
GV/Advanced Rendering			Y
GV/Computer Animation			Y
GV/Visualization			Y

GV/Fundamental Concepts

[2 Core-Tier1 and 1 Core-Tier2 hours]

For nearly every computer scientist and software developer, an understanding of how humans interact with machines is essential. While these topics may be covered in a standard undergraduate graphics course, they may also be covered in introductory computer science and programming courses. Part of our motivation for including immediate and retained modes is that these modes are roughly analogous to polling vs. event driven programming. This is a fundamental question in computer science: Is there a button object, or is there just the display of a button on the screen? Note that most of the outcomes in this section are at the knowledge level, and many of these topics may be revisited in greater depth.

Topics:

[Core-Tier1]

- Basics of Human visual perception (HCI Foundations).
- Image representations, vector vs. raster, color models, meshes.
- Forward and backward rendering (i.e., ray-casting and rasterization).
- Applications of computer graphics: including game engines, cad, visualization, virtual reality.

[Core-Tier2]

- Polygonal representation.
- Basic radiometry, similar triangles, and projection model.
- Use of standard graphics APIs (see HCI GUI construction).
- Compressed image representation and the relationship to information theory.
- Immediate and retained mode.
- Double buffering.

Learning Outcomes:

Students should be able to:

1. Describe the basic process of human visual perception including the perception of continuous motion from a sequence of discrete frames (sometimes called “flicker fusion”), tricolor stimulus, depth cues, contrast sensitivity, and the limits of human visual acuity. [knowledge level]
2. Describe color models and their use in graphics display devices. [knowledge level]
3. Differentiate between vector and raster rendering. [knowledge level]
4. Introduce the algorithmic distinction between projecting light from surfaces forward to the screen (e.g., triangle rasterization and splatting) vs. tracing the path of light backwards (e.g., ray or beam tracing). [knowledge level]
5. Identify common uses of computer graphics. [knowledge level]
6. Model simple graphics images. [application level]
7. Derive linear perspective from similar triangles by converting points (x, y, z) to points (x/z, y/z, 1). [application level]
8. Create 2D or 3D images using a standard graphics API. [application level]
9. Describe the basic graphics pipeline and how forward and backward rendering factor in this. [knowledge level]
10. Describe the differences between lossy and lossless image compression techniques, for example as reflected in common graphics image file formats such as JPG, PNG, and GIF. [knowledge level]
11. Apply a data compression algorithm such as run-length, Haar-wavelet, JPEG encoding, Huffman coding or Ziv-Lempel. [application level]
12. Apply double-buffering in the generation of a graphics application. [application level]

GV/Basic Rendering

[Elective]

This section describes basic rendering and fundamental graphics techniques that nearly every undergraduate course in graphics will cover and that is essential for further study in graphics. Sampling and anti-aliasing is related to the effect of digitization and appears in other areas of computing, for example, in audio sampling.

Topics:

- Rendering in nature, i.e., the emission and scattering of light and its relation to numerical integration.
- Affine and coordinate system transformations.
- Ray tracing.
- Visibility and occlusion, including solutions to this problem such as depth buffering, Paiter's algorithm, and ray tracing.
- The forward and backward rendering equation.
- Simple triangle rasterization.
- Rendering with a shader-based API.
- Texture mapping, including minification and magnification (e.g., trilinear MIP-mapping).
- Application of spatial data structures to rendering.
- Sampling and anti-aliasing.
- Scene graphs and the graphics pipeline.

Learning Outcomes:

Students should be able to:

1. Discuss the light transport problem and its relation to numerical integration i.e., light is emitted, scatters around the scene, and is measured by the eye; the form is an integral equation without analytic solution, but we can approach it as numerical integration.
2. Obtain 2-dimensional and 3-dimensional points by applying affine transformations.
3. Apply 3-dimensional coordinate system and the changes required to extend 2D transformation operations to handle transformations in 3D.
4. Contrast forward and backward rendering.
5. Explain the concept and applications of texture mapping, sampling, and anti-aliasing.
6. Explain the ray tracing – rasterization duality for the visibility problem.
7. Implement simple procedures that perform transformation and clipping operations on simple 2-dimensional images.
8. Implement a simple real-time renderer using a rasterization API (e.g., OpenGL) using vertex buffers and shaders.
9. Compare and contrast the different rendering techniques.
10. Compute space requirements based on resolution and color coding.
11. Compute time requirements based on refresh rates, rasterization techniques.

GV/Geometric Modeling

[Elective]

Topics:

- Basic geometric operations such as intersection calculation and proximity tests
- Volumes, voxels, and point-based representations.
- Parametric polynomial curves and surfaces.
- Implicit representation of curves and surfaces.
- Approximation techniques such as polynomial curves, Bezier curves, spline curves and surfaces, and non-uniform rational basis (NURB) spines, and level set method.
- Surface representation techniques including tessellation, mesh representation, mesh fairing, and mesh generation techniques such as Delaunay triangulation, marching cubes, .
- Spatial subdivision techniques.
- Procedural models such as fractals, generative modeling, and L-systems.
- Graftals, cross referenced with programming languages (grammars to generated pictures).
- Elastically deformable and freeform deformable models.
- Subdivision surfaces.
- Multiresolution modeling.
- Reconstruction.
- Constructive Solid Geometry (CSG) representation.

Learning Outcomes:

1. Represent curves and surfaces using both implicit and parametric forms.
2. Create simple polyhedral models by surface tessellation.
3. Implement such algorithms as
4. Generate a mesh representation from an implicit surface.
5. Generate a fractal model or terrain using a procedural method.
6. Generate a mesh from data points acquired with a laser scanner.
7. Construct CSG models from simple primitives, such as cubes and quadric surfaces.
8. Contrast modeling approaches with respect to space and time complexity and quality of image.

GV/Advanced Rendering

[Elective]

Topics:

- Solutions and approximations to the rendering equation, for example:
- Distribution ray tracing and path tracing
- Photon mapping
- Bidirectional path tracing
- Reyes (micropolygon) rendering
- Metropolis light transport
- Considering the dimensions of time (motion blur), lens position (focus), and continuous frequency (color).
- Shadow mapping.
- Occlusion culling.
- Bidirectional Scattering Distribution function (BSDF) theory and microfacets.
- Subsurface scattering.
- Area light sources.
- Hierarchical depth buffering.

- The Light Field, image-based rendering.
- Non-photorealistic rendering.
- GPU architecture.
- Human visual systems including adaptation to light, sensitivity to noise, and flicker fusion.

Learning Outcomes:

1. Demonstrate how an algorithm estimates a solution to the rendering equation.
2. Prove the properties of a rendering algorithm, e.g., complete, consistent, and/or unbiased.
3. Analyze the bandwidth and computation demands of a simple algorithm.
4. Implement a non-trivial shading algorithm (e.g., toon shading, cascaded shadow maps) under a rasterization API.
5. Discuss how a particular artistic technique might be implemented in a renderer.
6. Explain how to recognize the graphics techniques used to create a particular image.
7. Implement any of the specified graphics techniques using a primitive graphics system at the individual pixel level.
8. Implement a ray tracer for scenes using a simple (e.g., Phong's) BRDF plus reflection and refraction.

GV/Computer Animation

[Elective]

Topics:

- Forward and inverse kinematics.
- Collision detection and response
- Procedural animation using noise, rules (boids/crowds), and particle systems.
- Skinning algorithms.
- Physics based motions including rigid body dynamics, physical particle systems, mass-spring networks for cloth and flesh and hair.
- Key-frame animation.
- Splines.
- Data structures for rotations, such as quaternions.
- Camera animation.
- Motion capture.

Learning Outcomes:

1. Compute the location and orientation of model parts using an forward kinematic approach.
2. Compute the orientation of articulated parts of a model from a location and orientation using an inverse kinematic approach.
3. Describe the tradeoffs in different representations of rotations.
4. Implement the spline interpolation method for producing in-between positions and orientations.
5. Implement algorithms for physical modeling of particle dynamics using simple Newtonian mechanics, for example Witkin & Kass, snakes and worms, symplectic Euler, Stormer/Verlet, or midpoint Euler methods.
6. Describe the tradeoffs in different approaches to ODE integration for particle modeling.
7. Discuss the basic ideas behind some methods for fluid dynamics for modeling ballistic trajectories, for example for splashes, dust, fire, or smoke.
8. Use common animation software to construct simple organic forms using metaball and skeleton.

GV/Visualization

[Elective]

Visualization has strong ties to Human Computer Interaction as well as Computational Science. Readers should refer to the HCI and CN KAs for additional topics related to user population and interface evaluations.

Topics:

- Visualization of 2D/3D scalar fields: color mapping, isosurfaces.
- Direct volume data rendering: ray-casting, transfer functions, segmentation.
- Visualization of:
- Vector fields and flow data
- Time-varying data
- High-dimensional data: dimension reduction, parallel coordinates,
- Non-spatial data: multi-variate, tree/graph structured, text
- Perceptual and cognitive foundations that drive visual abstractions.
- Visualization design.
- Evaluation of visualization methods.
- Applications of visualization.

Learning Outcomes:

1. Describe the basic algorithms for scalar and vector visualization.
2. Describe the tradeoffs of algorithms in terms of accuracy and performance.
3. Propose a suitable visualization design for a particular combination of data characteristics and application tasks.
4. Discuss the effectiveness of a given visualization for a particular task.
5. Design a process to evaluate the utility of a visualization algorithm or system.
6. Recognize a variety of applications of visualization including representations of scientific, medical, and mathematical data; flow visualization; and spatial analysis.

Human-Computer Interaction (HC)

Human–computer interaction (HCI) is concerned with designing the interaction between people and computers and the construction of interfaces to afford this interaction.

Interaction between users and computational artifacts occurs at an interface which includes both software and hardware. Interface design impacts the software life-cycle in that it should occur early; the design and implementation of core functionality can influence the user interface – for better or worse.

Because it deals with people as well as computers, as a knowledge area HCI draws on a variety of disciplinary traditions including psychology, computer science, product design, anthropology and engineering.

HC: Human Computer Interaction (4 Core-Tier1 hours, 4 Core-Tier2 hours)

	Core-Tier1 hours	**Core-Tier2 hours**	**Includes Electives**
HC/Foundations	**4**		**N**
HC/Designing Interaction		**4**	**N**
HC/Programming Interactive Systems	**Elective**		
HC/User-cantered design & testing	**Elective**		
HC/Design for non-Mouse interfaces	**Elective**		
HC/Collaboration & communication	**Elective**		
HC/Statistical Methods for HCI	**Elective**		
HC/Human factors & security	**Elective**		
HC/Design-oriented HCI	**Elective**		
HC/Mixed, Augmented and Virtual Reality	**Elective**		

HC/Foundations

[4 Core-Tier1 hours]

Motivation: For end-users, the interface is the system. So design in this domain must be interaction-focused and human-centered. Students need a different repertoire of techniques to address interaction design than is provided elsewhere in the curriculum.

Topics:

- Contexts for HCI (anything with a user interface: webpage, business applications, mobile applications, games, etc.)
- Processes for user-centered development: early focus on users, empirical testing, iterative design.
- Different measures for evaluation: utility, efficiency, learnability, user satisfaction.
- Physical capabilities that inform interaction design: color perception, ergonomics
- Cognitive models that inform interaction design: attention, perception and recognition, movement, and memory. Gulfs of expectation and execution.
- Social models that inform interaction design: culture, communication, networks and organizations.
- Principles of good design and good designers; engineering tradeoffs
- Accessibility: interfaces for differently-abled populations (e.g. blind, motion-impaired)
- Interfaces for differently-aged population groups (e.g. children, 80+)

Learning Outcomes:

Students should be able to:

1. Discuss why human-centered software development is important (knowledge)
2. Summarize the basic precepts of psychological and social interaction (knowledge)
3. Develop and use a conceptual vocabulary for analyzing human interaction with software: affordance, conceptual model, feedback, and so forth (comprehension)
4. Define a user-centered design process that explicitly recognizes that the user is not like the developer or her acquaintances (comprehension)
5. Create and conduct a simple usability test for an existing software application (application)

HC/Designing Interaction

[4 Core-Tier2 hours]

Motivation: CS students need a minimal set of well-established methods and tools to bring to interface construction.

Topics:

- Principles of different styles of interface: e.g. command line, graphical tangible.
- Basic two-dimensional design fundamentals as applied to the visual interface, including use of grid, typography, color and contrast, scale, ordering and hierarchy.)
- Task analysis
- Paper prototyping
- Basic statistics and techniques for controlled experimentation (especially in regard to web data)
- KLM evaluation
- Help & documentation
- Handling human/system failure
- User interface standards

Learning Outcomes:

1. Students should be able to apply the principles of HCI foundations to:
2. Create a simple application, together with help & documentation, that supports a user interface (application)
3. Conduct a quantitative evaluation and discuss/report the results (application)
4. Discuss at least one national or international user interface design standard (comprehension)

HC/Programming Interactive Systems

[Elective]

Motivation: To take a user-experience-centered view of software development and then cover approaches and technologies to make that happen.

Topics:

Software Architecture Patterns: Model-View controller; command objects, online, offline, (cross reference SE/Software Design)

- Interaction Design Patterns: visual hierarchy, navigational distance
- Event management and user interaction
- Geometry management (cross reference GV/Geometric Modeling*)*
- Choosing interaction styles and interaction techniques
- Presenting information: navigation, representation, manipulation
- Interface animation techniques (scene graphs, etc)
- Widget classes and libraries
- Modern GUI libraries (iOS, Android, JavaFX) GUI builders and UI programming environments (cross reference to PBD/Mobile Platforms)
- Declarative Interface Specification: Stylesheets and DOMs
- Data-driven applications (database-backed web pages)
- Cross-platform design
- Design for resource-constrained devices (e.g. small, mobile devices)

Learning Outcomes:

Students should be able to apply the principles of HCI foundations to:

1. Understand there are common approaches to design problems, and be able to explain the importance of MVC to GUI programming (knowledge)
2. Create an application with a modern l user interface (application)
3. Identify commonalities and differences in UIs across different platforms (application)
4. Explain and use GUI programming concepts: event handling, constraint-based layout management, etc (evaluation)

HC/User-centered design and testing [elective]

Motivation: An exploration of techniques to ensure that end-users are fully considered at all stages of the design process, from inception to implementation.

Topics:

- Approaches and characteristics of design process
- Functionality and usability requirements (cross reference to SE Software Design)
- Techniques for gathering requirements: interviews, surveys, ethnographic & contextual enquiry (cross reference to SE Requirements Engineering)
- Techniques and tools for analysis & presentation of requirements: reports, personas
- Prototyping techniques and tools: sketching, storyboards, low-fidelity prototyping, wireframes
- Evaluation without users, using both qualitative and quantitative techniques: walkthroughs, GOMS, expert-based analysis, heuristics, guidelines, and standards
- Evaluation with users: observation, think-aloud, interview, survey, experiment.
- Challenges to effective evaluation: sampling, generalization.
- Reporting the results of evaluations
- Internationalization, designing for users from other cultures, cross-cultural evaluation

Learning Outcomes:

Students should be able to apply the principles of HCI foundations to:

1. Understand how user-centered design complements other software process models (knowledge)
2. Choose appropriate methods to support the development of a specific UI (application)
3. Use a variety of techniques to evaluate a given UI (application)
4. Use lo-fi prototyping techniques to gather, and report, user responses (application)
5. Describe the constraints and benefits of different evaluative methods (comprehension)

HC/Design for non-mouse interfaces

[Elective]

Motivation: As technologies evolve, new interaction styles are made possible. This knowledge unit should be considered extensible, to track emergent technology.

Topics:

- Choosing interaction styles and interaction techniques
- Representing information to users: navigation, representation, manipulation
- Approaches to design, implementation and evaluation of non-mouse interaction
- Touch and multi-touch interfaces
- New Windows (iPhone, Android)
- Speech recognition and natural language processing – (cross reference IS/Perception and Computer Vision)
- Wearable and tangible interfaces
- Persuasive interaction and emotion
- Ubiquitous and context-aware (Ubicomp)
- Bayesian inference (e.g. predictive text, guided pointing)
- Ambient/peripheral display and interaction

Learning Outcomes:

Students should be able to apply the principles of HCI foundations to:

1. Describe when non-mouse interfaces are appropriate (knowledge)
2. Discuss the advantages (and disadvantages) of non-mouse interfaces (application)
3. Understand the interaction possibilities beyond mouse-and-pointer interfaces (comprehension)

HC/Collaboration and communication

[Elective]

Motivation: Computer interfaces not only support users in achieving their individual goals but also in their interaction with others, whether that is task-focused (work or gaming) or task-unfocussed (social networking).

Topics:

- Asynchronous group communication: e-mail, forums, Facebook
- Synchronous group communication: chat rooms, conferencing, online games
- Online communities
- Software characters and intelligent agents, virtual worlds and avatars (cross reference IS/Agents)
- Social psychology
- Social networking
- Social computing

Learning Outcomes:

Students should be able to apply the principles of HCI foundations to:

1. Describe the difference between synchronous and asynchronous communication (knowledge)
2. Compare the HCI issues in individual interaction with group interaction (comprehension)
3. Discuss several issues of social concern raised by collaborative software (comprehension)
4. Discuss the HCI issues in software that embodies human intention (comprehension)

HC/Statistical methods for HCI

[Elective]

Motivation: Much HCI work depends on the proper use, understanding and application of statistics. This knowledge is often held by students who join the field from psychology, but less common in students with a CS background.

Topics:

- t-tests
- ANOVA
- randomization (non-parametric) testing, within v. between-subjects design
- calculating effect size
- exploratory data analysis
- presenting statistical data

- using statistical data
- using qualitative and quantitative results together

Learning Outcomes:

Students should be able to apply the principles of HCI foundations to:

1. Explain basic statistical concepts and their areas of application (knowledge)
2. Extract and articulate the statistical arguments used in papers which report HCI results (comprehension)
3. Devise appropriate statistical tests for a given HCI problem (application)

HC/Human factors and security

[Elective]

Motivation: Effective interface design requires basic knowledge of security psychology. Many attacks do not have a technological basis, but exploit human propensities and vulnerabilities. “Only amateurs attack machines; professionals target people” (Bruce Schneier)

Topics:

- Applied psychology and security policies
- Security economics
- Regulatory environments – responsibility, liability and self-determination
- Organizational vulnerabilities and threats
- Usability design and security
- Pretext, impersonation and fraud. Phishing and spear phishing (cross reference IAS/Fundamentals)
- Trust, privacy and deception
- Biometric authentication (camera, voice)
- Identity management

Learning Outcomes:

Students should be able to apply the principles of HCI foundations to:

1. Explain the concepts of phishing and spear phishing, and how to recognize them (knowledge)
2. Explain the concept of identity management and its importance (knowledge)
3. Describe the issues of trust in interface design with an example of a high and low trust system (knowledge)
4. Design a user interface for a security mechanism (application)
5. Analyze a security policy and/or procedures to show where they consider, or fail to consider, human factors (comprehension)

HC/Design-oriented HCI

[Elective]

Motivation: Some curricula will want to emphasize an understanding of the norms and values of HCI work itself as emerging from, and deployed within specific historical, disciplinary and cultural contexts.

Topics:

- Intellectual styles and perspectives to technology and its interfaces
- Consideration of HCI as a design discipline:
- Sketching
- Participatory design
- Critically reflective HCI
- Critical technical practice
- Technologies for political activism
- Philosophy of user experience
- Ethnography and ethno-methodology
- Indicative domains of application
- Sustainability
- Arts-informed computing

Learning Objectives

Students should be able to apply the principles of HCI foundations to:

1. Detail the processes of design appropriate to specific design orientations (knowledge)
2. Apply a variety of design methods to a given problem (application)
3. Understand HCI as a design-oriented discipline. (comprehension)

HC/Mixed, Augmented and Virtual Reality

[Elective]

Motivation: A detailed consideration of the interface components required for the creation and development of immersive environments, especially games.

Topics:

- Output
- Sound
- Stereoscopic display
- Force feedback simulation, haptic devices
- User input
- Viewer and object tracking
- Pose and gesture recognition
- Accelerometers
- Fiducial markers
- User interface issues
- Physical modeling and rendering
- Physical simulation: collision detection & response, animation

- Visibility computation
- Time-critical rendering, multiple levels of details (LOD)
- System architectures
- Game engines
- Mobile augmented reality
- Flight simulators
- CAVEs
- Medical imaging
- Networking
- p2p, client-server, dead reckoning, encryption, synchronization
- Distributed collaboration

Learning Objectives:

1. Describe the optical model realized by a computer graphics system to synthesize stereoscopic view (knowledge)
2. Describe the principles of different viewer tracking technologies. (knowledge)
3. Describe the differences between geometry- and image-based virtual reality.(knowledge)
4. Describe the issues of user action synchronization and data consistency in a networked environment.(knowledge)
5. Determine the basic requirements on interface, hardware, and software configurations of a VR system for a specified application. (application)
6. To be aware of the range of possibilities for games engines, including their potential and their limitations. (comprehension)

Information Assurance and Security (IAS)

In CS2013, the Information Assurance and Security KA is added to the Body of Knowledge in recognition of the world's reliance on information technology and its critical role in computer science education. Information assurance and security as a domain is the set of controls and processes both technical and policy intended to protect and defend information and information systems by ensuring their availability, integrity, authentication, and confidentiality and providing for non-repudiation. The concept of assurance also carries an attestation that current and past processes and data are valid. Both assurance and security concepts are needed to ensure a complete perspective. Information assurance and security education, then, includes all efforts to prepare a workforce with the needed knowledge, skills, and abilities to protect our information systems and attest to the assurance of the past and current state of processes and data. The Information Assurance and Security KA is unique among the set of KA's presented here given the manner in which the topics are pervasive throughout other Knowledge Areas. The topics germane to only IAS are presented in depth in the IAS section; other topics are noted and cross referenced in the IAS KA, with the details presented in the KA in which they are tightly integrated.

The aim of this KA is two-fold. First, the KA defines the core (tier1and tier2) and the elective components that depict topics that are part of an undergraduate computer science curriculum. Secondly (and almost more importantly), we document the pervasive presence of IAS within a computer science undergraduate curriculum.

The IAS KA is shown in two groups; (1) concepts that are, at the first order, germane to Information Assurance and Security and (2) IAS topics that are integrated into other KA's. For completeness, the total distribution of hours is summarized in the table below.

	Core-Tier1 hours	**Core-Tier2 hours**	**Elective Topics**
IAS	**2**	**6**	**Y**
IAS distributed in other KA's	**23**	**46**	**Y**

IAS. Information Assurance and Security (2 Core-Tier1 hours, 6 Core-Tier2 hours)

	Core-Tier1 hours	Core-Tier2 hours	Includes Electives
IAS/Fundamental Concepts	1	2	N
IAS/Network Security	1	4	N
IAS/Cryptography			Y
IAS/Risk Management			Y
IAS/Security Policy and Governance			Y
IAS / Digital Forensics			Y
IAS / Security Architecture and Systems Administration			Y
IAS/Secure Software Design and Engineering			Y

IAS. Information Assurance and Security (distributed) (23 Core-Tier1 hours, 46 Core-Tier2 hours)

Knowledge Area and Topic	Core-Tier1 hours	Core-Tier2 hours	Includes Electives
OS/ Overview of OS	1*		
OS/OS Principles	1*		
OS/Concurrency		3	
OS/Scheduling and Dispatch		3	
OS/Memory Management		1*	
OS/Security and Protection		2	
OS/Virtual Machines			Y
OS/Device Management			Y
OS/File Systems			Y
OS/Real Time and Embedded Systems			Y

OS/Fault Tolerance			**Y**
OS/System Performance Evaluation			**Y**
NC/Introduction	**1.5**		
NC/Networked Applications	**1.5**		
NC/Reliable Data Delivery		**2**	
NC/Routing and Forwarding		**1.5**	
NC/Local Area Networks		**1.5**	
NC/Resource Allocation		**1**	
NC/Mobility		**1**	
PBD/Web Platforms			**Y**
PBD/Mobile Platforms			**Y**
PBD/Industrial Platforms			**Y**
IM/Information Management Concepts		**2**	
IM/Transaction Processing			**Y**
IM/Distributed Databases			**Y**
PL/Functional Programming		**2**	
PL/Type Systems	**1**	**4**	
PL/Language Translation And Execution	**1**	**3**	
PD/Parallelism Fundamentals	**1***		**Y**
PD/Communication and Coordination	**1**	**3**	
SDF/Development Methods	**9**		

SE/Software Processes	**1**		
SE/Software Project Management		**3**	
SE/Tools and Environments		**1**	
SE/Software Construction		**2**	**Y**
SE/Software Verification Validation		**3**	**Y**
SP/Professional Ethics	**2**	**1**	
SP/Intellectual Property	**2**		
SP/Security Policies, Laws and Computer Crimes			**Y**
HCI/Human factors and security			**Y**
IS/Reasoning Under Uncertainty			**Y**

* Indicates not all hours in the KU are classified as cross referenced to IAS

IAS/Fundamental Concepts

[1 Core-Tier1 hours, 2 Core-Tier2 hours]

Topics:

[Core-Tier1]

- Nature of the Threats
- Need for Information Assurance.
- Basic Terminology that should be recognized by those studying the field. (Confidentiality, Integrity, Availability)
- Information Assurance Concepts that are key to building an understanding of the IA area.

[Core-Tier2]

- Industry and Government Guidelines and Standards concerning Information Assurance.
- National and Cultural Differences including topics such as HIPAA, Safe Harbor, and data protection laws.
- Legal, Ethical, and Social Issues (cross reference with SP KA)
- Threats and Vulnerabilities.
- Types of Attacks
- Types of Attackers.
- Defense Mechanisms.
- Incident Response.

Learning outcomes:

1. Describe the types of threats to data and information systems [Knowledge]
2. Describe why processes and data need protection [Knowledge]
3. Describe the context in which Confidentiality, Integrity, and Availability are important to given processes or data? [Application]
4. Determine if the security controls provide sufficient security for the required level of Confidentiality, Integrity, and/or Availability [Evaluation]
5. What are significant national level laws affecting the obligation for the protection of data? [Knowledge]
6. Describe how laws affecting privacy and data/IP protection differ based on country? [Evaluation]
7. Describe the major vulnerabilities present in systems today. [Knowledge]
8. Define the fundamental motivations for intentional malicious exploitation of vulnerabilities. [Knowledge]
9. Define the defense mechanisms that can be used to detect or mitigate malicious activity in IT systems. [Knowledge]
10. Define an incident. [Knowledge]
11. Enumerate the roles required in incident response and the common steps after an incident has been declared. [Knowledge]
12. Describe the actions taken in response to the discovery of a given incident. [Application]

IAS/Network Security

[1 Core-Tier1 hours, 4 Core-Tier2 hours]

Discussion of network security relies on previous understanding on fundamental concepts of networking, including protocols, such as TCP/IP, and network architecture/organization (xref NC/Network Communication).

Topics:

[Core-Tier1]

- Application of Cryptography
- TLS
- Secret-key algorithms
- Public-key algorithms
- Hybrid

[Core-Tier2]

- Network attack types: Denial of service, flooding, sniffing and traffic redirection, message integrity attacks,
- Identity hijacking, exploit attacks (buffer overruns, Trojans, backdoors), inside attacks, infrastructure (DNS hijacking, route blackholing, misbehaving routers that drop traffic), etc.)
- Authentication protocols
- Digital signatures
- Message Digest
- Defense Mechanisms /Countermeasures. (Intrusion Detection, Firewalls, Detection of malware, IPSec, Virtual Private Networks, Network Address Translation.)
- Network Auditing.

Learning outcomes:

1. Identify protocols used to enhance Internet communication, and choose the appropriate protocol for a particular [Knowledge]
2. Discuss the difference between secret key and public key encryption. [Knowledge]

3. Discuss the fundamental ideas of public-key cryptography. [Knowledge]
4. Discuss the role of a certificate authority in public-key cryptography. [Knowledge]
5. Discuss non-repudiation [Knowledge]
6. Describe a digital signature[Knowledge]
7. Describe how public key encryption is used to encrypt email traffic. [Knowledge]
8. Generate and distribute a PGP key pair and use the PGP package to send an encrypted e-mail message. [Application]
9. Describe how public key encryption is used to secure HTTP traffic. [Knowledge]
10. Describe the security risks present in networking. [Knowledge]
11. Discuss the differences in Network Intrusion Detection and Network Intrusion Prevention. [Knowledge]
12. Describe how the basic security implications of a hub and a switch. [Knowledge]
13. Describe how a system can intercept traffic in a local subnet. [Knowledge]
14. Describe different implementations for intrusion detection. [Knowledge]
15. Identify a buffer overflow vulnerability in code [Evaluation]
16. Correct a buffer overflow error in code [Application]
17. Describe the methods that can be used to alert that a system has a backdoor installed. [Knowledge]
18. Describe the methods that can be used to identify a system is running processes not desired be the system owner. [Knowledge]
19. Analyze a port listing for unwanted TCP/UDP listeners. [Application]
20. Describe the difference between non-routable and routable IP addresses. [Knowledge]
21. List the class A, B, and C non-routable IP ranges. [Knowledge]
22. Describe the difference between stateful and non-stateful firewalls. [Knowledge]
23. Implement firewalls to prevent specific IP’s or ports from traversing the firewall. [Application]
24. Describe the different actions a firewall can take with a packet. [Knowledge]
25. Summarize common authentication protocols. [Knowledge]
26. Describe and discuss recent successful security attacks. [Knowledge]
27. Summarize the strengths and weaknesses associated with different approaches to security. [Knowledge]
28. Describe what a message digest is and how it is commonly used. [Knowledge]

IAS/ Cryptography

[Elective]

Topics:

- The Basic Cryptography Terminology covers notions pertaining to the different (communication) partners, secure/unsecure channel, attackers and their capabilities, encryption, decryption, keys and their characteristics, signatures, etc.
- Cipher types:, Caesar cipher, affine cipher, etc. together with typical attack methods such as frequency analysis, etc.
- Mathematical Preliminaries; include topics in linear algebra, number theory, probability theory, and statistics. (Discrete Structures)
- Cryptographic Primitives include encryption (stream ciphers, block ciphers public key encryption), digital signatures, message authentication codes, and hash functions.
- Cryptanalysis covers the state-of-the-art methods including differential cryptanalysis, linear cryptanalysis, factoring, solving discrete logarithm problem, lattice based methods, etc.
- Cryptographic Algorithm Design covers principles that govern the design of the various cryptographic primitives, especially block ciphers and hash functions. (Algorithms and Complexity - Hash functions)
- The treatment of Common Protocols includes (but should not be limited to) current protocols such as RSA, DES, DSA, AES, ElGamal, MD5, SHA-1, Diffie-Hellman Key exchange, identification and authentication protocols, secret sharing, multi-party computation, etc.
- Public Key Infrastructure deals with challenges, opportunities, local infrastructures, and national infrastructure.

Learning outcomes:

1. What is the purpose of Cryptography? [Knowledge]
2. What is plain text? [Knowledge]
3. What is cipher text? [Knowledge]
4. What are the two basic methods (ciphers) for transforming plain text in cipher text? [Knowledge]
5. Describe attacks against a specified cypher. [Knowledge]
6. Define the following terms: Cipher, Cryptanalysis, Cryptographic Algorithm, Cryptology. [Knowledge]
7. What is the Work Function of a given cryptographic algorithm? [Knowledge]
8. What is a One Time Pad (Vernam Cipher)? [Knowledge]
9. What is a Symmetric Key operation? [Knowledge]
10. What is an Asymmetric Key operation? [Knowledge]
11. For a given problem and environment weigh the tradeoffs between a Symmetric and Asymmetric key operation. [Evaluation]
12. What are common Symmetric Key algorithms? [Knowledge]
13. Explain in general how a public key algorithm works. [Knowledge]
14. How does “key recovery” work? [Knowledge]
15. List 5 public key algorithms. [Knowledge]
16. Describe the process in the Diffie-Hellman key exchange. [Knowledge]
17. What is a message digest and list 4 common algorithms? [Knowledge]
18. What is a digital signature and how is one created? [Knowledge]
19. What the three components of a PKI? [Knowledge]
20. List the ways a PKI infrastructure can be attacked. [Knowledge]

IAS/Risk Management

[Elective]

Topics:

- Risk Analysis involves identifying the assets, probable threats, vulnerabilities and control measures to discern risk levels and likelihoods. It can be applied to a program, organization, sector, etc. Knowledge in this area includes knowing different risk analysis models and methods, their strengths and benefits and the apropriateness of the different methods and models given the situation. This includes periodic reassessment.
- Cost/Benefit Analysis is used to weigh private and/or public costs versus benefits and can be applied to security policies, investments, programs, tools, deployments, etc.
- Continuity Planning will help organizations deliver critical services and ensure survival.
- Disaster Recovery will help an organization continue normal operations in a minimum amount of time with a minimum amount of disruption and cost.
- Security Auditing: a systematic assessment of an organization’s system measuring the conformity vis-àvis a set of pre-established criteria.
- Asset Management minimizes the life cost of assets and includes critical factors such as risk or business continuity.
- Risk communication Enforcement of risk management policies is critical for an organization.

Learning outcomes:

1. How is risk determined? [Knowledge]
2. What does it mean to manage risk? [Knowledge]
3. What is the primary purpose of risk management? [Knowledge]
4. Who can accept Risk? [Knowledge]
5. What is the objective of Security Controls in security management? [Knowledge]
6. With respect to a risk program, what is an Asset? [Knowledge]
7. With respect to a risk program, what is a Threat? [Knowledge]

8. With respect to a risk program, what is a Vulnerability? [Knowledge]
9. With respect to a risk program, what is a Safeguard? [Knowledge]
10. With respect to a risk program, what is the Exposure Factor (EF)? [Knowledge]
11. What is the difference between Quantitative Risk Analysis and Qualitative Risk Analysis? [Knowledge]
12. How does an organization determine what safeguards or controls to implement? [Knowledge]
13. Given the value of an asset and the cost of the security controls to mitigate loss/damage/destruction, is the security plan appropriate? [Evaluation]
14. What is Risk Analysis (RA)? [Knowledge]
15. Describe how data is classified in either (government or commercial)? [Knowledge]
16. When are the factors used when determining the classification of a piece of information? [Knowledge]
17. What are three ways to deal with Risk? [Knowledge]

IAS/Security Policy and Governance

[Elective]

Topics:

- Strategies and Plans for creating security policies.
- Policies, Guidelines, Standards and Best Practices for individuals or organizations, including national security policies.
- Procedures for creating policies, guidelines, standards, specifications, regulations and laws.
- Privacy Policies to help protect personal and other sensitive information.
- Compliance and Enforcement of policies, standards, regulations, and laws.
- Formal Policy Models such as Bell-LaPadula, Biba and Clark-Wilson, which provide precise specifications of security objectives.
- Relation of national security policies, regulations, organizational security policies, formal policy models, and policy languages.
- Policy as related to Risk Aversion.

Learning outcomes:

1. What is a security policy and why does an organization need a security policy? [Knowledge]
2. Come up with an example of your own, which would be caused by missing security policies.[Application]
3. What are the basic things that need to be explained to every employee about a security policy? At what point in their employment? Why? [Application]
4. Say you have an e-mail server that processes sensitive emails from important people. What kind of things should be put into the security policy for the email server? [Evaluation]
5. Read your institution’s security plan and critique the plan. [Evaluation]
6. Update your institution’s security plan. [Evaluation]

IAS/ Digital Forensics

[Elective]

Topics:

- Basic Principles and methodologies for digital forensics.
- Rules of Evidence – general concepts and differences between jurisdictions and Chain of Custody.
- Search and Seizure of evidence, e.g., computers, including search warrant issues.
- Digital Evidence methods and standards.
- Techniques and standards for Preservation of Data.

- Data analysis and validation.
- Legal and Reporting Issues including working as an expert witness.
- OS/File System Forensics
- Application Forensics
- Network Forensics
- Mobile Device Forensics
- Computer/network/system attacks.

Learning outcomes:

1. What is a Digital Investigation? [Knowledge]
2. What systems in an IT infrastructure might have forensically recoverable data? [Knowledge]
3. Who in an organization is authorized to permit the conduct of a forensics investigation? [Knowledge]
4. What is the Rule of Evidence? [Knowledge]
5. What is a Chain of Custody? [Knowledge]
6. Conduct a data collection on a hard drive. [Application]
7. Validate the integrity of a digital forensics data set. [Application]
8. Determine if a digital investigation is sound. [Evaluation]
9. Describe the file system structure for a given device (NTFA, MFS, iNode, HFS…) [Knowledge]
10. Determine if a certain string of data exists on a hard drive. [Application]
11. Describe the capture of live data for a forensics investigation. [Knowledge]
12. Capture and interpret network traffic. [Application]
13. Discuss identity management and its role in access control systems. [Knowledge]
14. Determine what user was logged onto a given system at a given time. [Application]
15. Determine the submissability (from a legal perspective) of data. [Evaluation]
16. Evaluate a system for the presence of malware. [Evaluation]

IAS/Security Architecture and Systems Administration

[Elective]

Topics:

- How to secure Hardware, including how to make hardware tokens and chip cards tamper-proof and tamper-resistance.
- Configuring systems to operate securely as an IT system.
- Access Control
- Basic Principles of an access control system prevent unauthorized access.
- Physical Access Control determines who is allowed to enter or exit, where the user is allowed to enter or exit, and when the user is allowed to enter or exit.
- Technical/System Access Control is the process of preventing unauthorized users or services to utilize information systems.
- Usability includes the difficulty for humans to deal with security (e.g., remembering PINs), social engineering, phishing, and other similar attacks.
- Analyzing and identifying System Threats and Vulnerabilities
- Investigating Operating Systems Security for various systems.
- Multi-level/Multi-lateral Security
- Design and Testing for architectures and systems of different scale
- Penetration testing in the system setting
- Products available in the marketplace
- Supervisory Control and Data Acquisition (SCADA)
- SCADA system uses. Communications protocols supporting data acquisition
- Communications protocols supporting distributed control.
- Data Integrity
- Data Confidentiality

Learning outcomes:

1. Explain the need for software security and how software security is different from security features like access control or cryptography. [Knowledge]
2. Understand common threats to web applications and common vulnerabilities written by developers. [Knowledge]
3. Define least privilege. [Knowledge]
4. Define “Defense in Depth”. [Knowledge]
5. Define service isolation in the context of enterprise systems. [Knowledge]
6. Architect an enterprise system using the concept of service isolation. [Application]
7. Describe the methods to provide for access control and what enterprise services must exist. [Knowledge]
8. Discuss how user systems integrate into an enterprise environment. [Knowledge]
9. Discuss the risks client systems pose to an enterprise environment. [Knowledge]
10. Describe various methods to manage client systems. [Knowledge]
11. Create a risk model of a web application, ranking and detailing the risks to the system’s assets. [Application]
12. Construct, document, and analyze security requirements with abuse cases and constraints. [Application]
13. Apply secure design principles, such as least privilege, to the design of a web application. [Application]
14. Validate both the input and output of a web application. [Application]
15. Use cryptography appropriately, including SSL and certificate management. [Application]
16. Create a test plan and conduct thorough testing of web applications with appropriate software assistance. [Application]

IAS/Secure Software Design and Engineering

[Elective]

Fundamentals of secure coding practices covered in other knowledge areas, including SDF/SE/PL.

Topics:

- Building security into the Software Development Lifecycle
- Secure Design Principles and Patterns (Saltzer and Schroeder, etc)
- Secure Software Specification and Requirements deals with specifying what the program should and should not do, which can be done either using a requirements document or using a more formal mathematical specification.
- Secure Coding involves applying the correct balance of theory and practice to minimize vulnerabilities in code.
- Data validation
- Memory handling
- Crypto implementation
- Secure Testing is the process of testing that security requirements are met (including Static and Dynamic analysis).
- Program Verification and Simulation is the process of ensuring that a certain version of a certain implementation meets the required security goals, either by a mathematical proof or by simulation.

Learning outcomes:

1. Describe the Design Principles for Protection Mechanisms (Saltzer and Schroeder) [Knowledge]
2. Describe the Principles for Software Security (Viega and McGraw) [Knowledge]
3. Define Principles for a Secure Design (Morrie Gasser) [Knowledge]
4. Compare the principles for software and systems in the context of a software development effort. [Application]
5. Discuss the benefits and drawbacks of open-source vs proprietary software and security [Knowledge]
6. Integrate trustworthy development practices into an existing software development lifecycle [Application]
7. Integrate authenticating libraries, DLL, run-time [Application]
8. Identify a buffer overflow in a code sample [Knowledge]
9. Describe the difference between static and dynamic analysis. [Knowledge]
10. Conduct static analysis to determine the security posture of a given application. [Application]
11. Monitor the execution of a software (dynamic analysis) and discuss the observed process flows. [Application]
12. How is quality assurance conducted for software development? [Knowledge]
13. Participate in a code review focused on finding security bugs using static analysis tools. [Application]
14. Where does patch management fit in a software development project? [Knowledge]

Information Management (IM)

Information Management (IM) is primarily concerned with the capture, digitization, representation, organization, transformation, and presentation of information; algorithms for efficient and effective access and updating of stored information, data modeling and abstraction, and physical file storage techniques. The student needs to be able to develop conceptual and physical data models, determine what IM methods and techniques are appropriate for a given problem, and be able to select and implement an appropriate IM solution that addresses relevant design concerns including scalability, accessibility and usability.

IM. Information Management (1 Core-Tier1 hour; 9 Core-Tier2 hours)

	Core-Tier1 hours	Core-Tier2 hours	Includes Electives
IM/Information Management Concepts	1	2	N
IM/Database Systems		3	N
IM/Data Modeling		4	N
IM/Indexing			Y
IM/Relational Databases			Y
IM/Query Languages			Y
IM/Transaction Processing			Y
IM/Distributed Databases			Y
IM/Physical Database Design			Y
IM/Data Mining			Y
IM/Information Storage And Retrieval			Y

IM/Information Management Concepts

[1 Core-Tier1 hour; 2 Core-Tier2 hours]

Topics:

[Core-Tier1]

- Basic information storage and retrieval (IS&R) concepts
- Information capture and representation
- Supporting human needs: Searching, retrieving, linking, browsing, navigating

[Core-Tier2]

- Information management applications
- Declarative and navigational queries, use of links
- Analysis and indexing
- Quality issues: Reliability, scalability, efficiency, and effectiveness

Learning Outcomes:

1. Describe how humans gain access to information and data to support their needs [Knowledge]
2. Compare and contrast information with data and knowledge [Analysis]
3. Demonstrate uses of explicitly stored metadata/schema associated with data [Application]
4. Identify issues of data persistence to an organization [Knowledge]
5. Critique/defend a small- to medium-size information application with regard to its satisfying real user information needs [Evaluation]
6. Explain uses of declarative queries [Knowledge]
7. Give a declarative version for a navigational query [Knowledge]
8. Describe several technical solutions to the problems related to information privacy, integrity, security, and preservation [Knowledge]
9. Explain measures of efficiency (throughput, response time) and effectiveness (recall, precision) [Knowledge]
10. Describe approaches that scale up to globally networked systems [Knowledge]
11. Identify vulnerabilities and failure scenarios in common forms of information systems [Knowledge]

IM/Database Systems

[3 Core-Tier2 hours]

Topics:

[Core-Tier2]

- Approaches to and evolution of database systems
- Components of database systems
- DBMS functions
- Database architecture and data independence
- Use of a declarative query language
- Systems supporting structured and/or stream content

Learning Outcomes:

1. Explain the characteristics that distinguish the database approach from the traditional approach of programming with data files [Knowledge]
2. Cite the basic goals, functions, models, components, applications, and social impact of database systems [Knowledge]
3. Describe the components of a database system and give examples of their use [Knowledge]
4. Identify major DBMS functions and describe their role in a database system [Knowledge]
5. Explain the concept of data independence and its importance in a database system [Knowledge]
6. Use a declarative query language to elicit information from a database [Application]
7. Describe how various types of content cover the notions of structure and/or of stream (sequence), e.g., documents, multimedia, tables [Knowledge]

IM/Data Modeling

[4 Core-Tier2 hours]

Topics:

[Core-Tier2]

- Data modeling
- Conceptual models (e.g., entity-relationship and UML diagrams)
- Relational data model
- Object-oriented model
- Semi-structured data model (expressed using DTD or XML Schema, for example)

Learning Outcomes:

1. Categorize data models based on the types of concepts that they provide to describe the database structure—that is, conceptual data model, physical data model, and representational data model [Comprehension]
2. Describe the modeling concepts and notation of widely used modeling notation (e.g., ERD notation, and UML), including their use in data modeling [Knowledge]
3. Define the fundamental terminology used in the relational data model [Knowledge]
4. Describe the basic principles of the relational data model [Knowledge]
5. Apply the modeling concepts and notation of the relational data model [Application]
6. Describe the main concepts of the OO model such as object identity, type constructors, encapsulation, inheritance, polymorphism, and versioning [Knowledge]
7. Describe the differences between relational and semi-structured data models [Knowledge]
8. Give a semi-structured equivalent (e.g., in DTD or XML Schema) for a given relational schema [Application]

IM/Indexing

[Elective]

Topics:

- The impact of indexes on query performance
- The basic structure of an index; [Robert: Not sure if this warrants a topic by itself]
- Keeping a buffer of data in memory; [Robert: Why is this listed as a topic?]
- Creating indexes with SQL
- Indexing text
- Indexing the web (how search engines work)

Learning Outcomes:

1. Generate an index file for a collection of resources.
2. Explain the role of an inverted index in locating a document in a collection
3. Explain how stemming and stop words affect indexing
4. Identify appropriate indices for given relational schema and query set
5. Estimate time to retrieve information, when indices are used compared to when they are not used.

IM/Relational Databases

[Elective]

Topics:

Elective

- Mapping conceptual schema to a relational schema
- Entity and referential integrity
- Relational algebra and relational calculus
- Relational Database design
- Functional dependency
- Decomposition of a schema; lossless-join and dependency-preservation properties of a decomposition
- Candidate keys, superkeys, and closure of a set of attributes
- Normal forms (1NF, 2NF, 3NF, BCNF)
- Multi-valued dependency (4NF)
- Join dependency (PJNF, 5NF)
- Representation theory

Learning Outcomes:

1. Prepare a relational schema from a conceptual model developed using the entity- relationship model
2. Explain and demonstrate the concepts of entity integrity constraint and referential integrity constraint (including definition of the concept of a foreign key).
3. Demonstrate use of the relational algebra operations from mathematical set theory (union, intersection, difference, and Cartesian product) and the relational algebra operations developed specifically for relational databases (select (restrict), project, join, and division).
4. Demonstrate queries in the relational algebra.
5. Demonstrate queries in the tuple relational calculus.
6. Determine the functional dependency between two or more attributes that are a subset of a relation.
7. Connect constraints expressed as primary key and foreign key, with functional dependencies
8. Compute the closure of a set of attributes under given functional dependencies
9. Determine whether or not a set of attributes form a superkey and/or candidate key for a relation with given functional dependencies
10. Evaluate a proposed decomposition, to say whether or not it has lossless-join and dependency-preservation
11. Describe what is meant by 1NF, 2NF, 3NF, and BCNF.
12. Identify whether a relation is in 1NF, 2NF, 3NF, or BCNF.
13. Normalize a 1NF relation into a set of 3NF (or BCNF) relations and denormalize a relational schema.
14. Explain the impact of normalization on the efficiency of database operations, especially query optimization.
15. Describe what is a multivalued dependency and what type of constraints it specifies.
16. Explain why 4NF is useful in schema design.

IM/Query Languages

[Elective]

Topics:

- Overview of database languages
- SQL (data definition, query formulation, update sublanguage, constraints, integrity)
- QBE and 4th-generation environments
- Embedding non-procedural queries in a procedural language
- Introduction to Object Query Language
- Stored procedures

Learning Outcomes:

1. Create a relational database schema in SQL that incorporates key, entity integrity, and referential integrity constraints.
2. Demonstrate data definition in SQL and retrieving information from a database using the SQL SELECT statement.
3. Evaluate a set of query processing strategies and select the optimal strategy.
4. Create a non-procedural query by filling in templates of relations to construct an example of the desired query result.
5. Embed object-oriented queries into a stand-alone language such as C++ or Java (e.g., SELECT Col.Method() FROM Object).
6. Write a stored procedure that deals with parameters and has some control flow, to provide a given functionality

IM/Transaction Processing

[Elective]

Topics:

- Transactions
- Failure and recovery
- Concurrency control

Learning Outcomes:

1. Create a transaction by embedding SQL into an application program.
2. Explain the concept of implicit commits.
3. Describe the issues specific to efficient transaction execution.
4. Explain when and why rollback is needed and how logging assures proper rollback.
5. Explain the effect of different isolation levels on the concurrency control mechanisms.
6. Choose the proper isolation level for implementing a specified transaction protocol.

IM/Distributed Databases

[Elective]

Topics:

- Distributed data storage
- Distributed query processing
- Distributed transaction model
- Concurrency control
- Homogeneous and heterogeneous solutions
- Client-server distributed databases (cross-reference SF/Computational Paradigms)

Learning Outcomes:

1. Explain the techniques used for data fragmentation, replication, and allocation during the distributed database design process.
2. Evaluate simple strategies for executing a distributed query to select the strategy that minimizes the amount of data transfer.
3. Explain how the two-phase commit protocol is used to deal with committing a transaction that accesses databases stored on multiple nodes.
4. Describe distributed concurrency control based on the distinguished copy techniques and the voting method.
5. Describe the three levels of software in the client-server model.

IM/Physical Database Design

[Elective]

Topics:

- Storage and file structure
- Indexed files
- Hashed files
- Signature files
- B-trees
- Files with dense index
- Files with variable length records
- Database efficiency and tuning

Learning Outcomes:

1. Explain the concepts of records, record types, and files, as well as the different techniques for placing file records on disk.
2. Give examples of the application of primary, secondary, and clustering indexes.
3. Distinguish between a non-dense index and a dense index.
4. Implement dynamic multilevel indexes using B-trees.
5. Explain the theory and application of internal and external hashing techniques.
6. Use hashing to facilitate dynamic file expansion.
7. Describe the relationships among hashing, compression, and efficient database searches.
8. Evaluate costs and benefits of various hashing schemes.
9. Explain how physical database design affects database transaction efficiency.

IM/Data Mining

[Elective]

Topics:

- The usefulness of data mining
- Data mining algorithms
- Associative and sequential patterns
- Data clustering
- Market basket analysis
- Data cleaning
- Data visualization

Learning Outcomes:

1. Compare and contrast different conceptions of data mining as evidenced in both research and application.
2. Explain the role of finding associations in commercial market basket data.
3. Characterize the kinds of patterns that can be discovered by association rule mining.
4. Describe how to extend a relational system to find patterns using association rules.
5. Evaluate methodological issues underlying the effective application of data mining.
6. Identify and characterize sources of noise, redundancy, and outliers in presented data.
7. Identify mechanisms (on-line aggregation, anytime behavior, interactive visualization) to close the loop in the data mining process.
8. Describe why the various close-the-loop processes improve the effectiveness of data mining.

IM/Information Storage and Retrieval

[Elective]

Topics:

- Characters, strings, coding, text
- Documents, electronic publishing, markup, and markup languages
- Tries, inverted files, PAT trees, signature files, indexing
- Morphological analysis, stemming, phrases, stop lists
- Term frequency distributions, uncertainty, fuzziness, weighting
- Vector space, probabilistic, logical, and advanced models
- Information needs, relevance, evaluation, effectiveness
- Thesauri, ontologies, classification and categorization, metadata
- Bibliographic information, bibliometrics, citations
- Routing and (community) filtering
- Search and search strategy, multimedia search, information seeking behavior, user modeling, feedback
- Information summarization and visualization
- Integration of citation, keyword, classification scheme, and other terms
- Protocols and systems (including Z39.50, OPACs, WWW engines, research systems)
- Digital libraries
- Digitization, storage, interchange, digital objects, composites, and packages
- Metadata, cataloging, author submission
- Naming, repositories, archives
- Spaces (conceptual, geographical, 2/3D, VR)
- Architectures (agents, buses, wrappers/mediators), interoperability
- Services (searching, linking, browsing, and so forth)

- Intellectual property rights management, privacy, and protection (watermarking)
- Archiving and preservation, integrity

Learning Outcomes:

1. Explain basic information storage and retrieval concepts.
2. Describe what issues are specific to efficient information retrieval.
3. Give applications of alternative search strategies and explain why the particular search strategy is appropriate for the application.
4. Perform Internet-based research.
5. Design and implement a small to medium size information storage and retrieval system, or digital library.
6. Describe some of the technical solutions to the problems related to archiving and preserving information in a digital library.

Intelligent Systems (IS)

Artificial intelligence (AI) is the study of solutions for problems that are difficult or impractical to solve with traditional methods. It is used pervasively in support of everyday applications such as email, word-processing and search, as well as in the design and analysis of autonomous agents that perceive their environment and interact rationally with the environment.

The solutions rely on a broad set of general and specialized knowledge representation schemes, problem solving mechanisms and learning techniques. They deal with sensing (e.g., speech recognition, natural language understanding, computer vision), problem-solving (e.g., search, planning), and acting (e.g., robotics) and the architectures needed to support them (e.g,. agents, multi-agents). The study of Artificial Intelligence prepares the student to determine when an AI approach is appropriate for a given problem, identify the appropriate representation and reasoning mechanism, and implement and evaluate it.

IS. Intelligent Systems (10 Core-Tier2 hours)

	Core-Tier1 hours	**Core-Tier2 hours**	**Includes Electives**
IS/Fundamental Issues		**1**	**Y**
IS/Basic Search Strategies		**4**	**N**
IS/Basic Knowledge Representation and Reasoning		**3**	**N**
IS/Basic Machine Learning		**2**	**N**
IS/Advanced Search			**Y**
IS/Advanced Representation and Reasoning			**Y**
IS/Reasoning Under Uncertainty			**Y**
IS/Agents			**Y**
IS/Natural Language Processing			**Y**
IS/Advanced Machine Learning			**Y**
IS/Robotics			**Y**
IS/Perception and Computer Vision			**Y**

IS/Fundamental Issues

[1 Core-Tier2 hours]

Topics:

- Overview of AI problems, Examples of successful recent AI applications
- What is intelligent behavior?
- The Turing test
- Rational versus non-rational reasoning
- Nature of human reasoning
- Nature of environments
- Fully versus partially observable
- Single versus multi-agent
- Deterministic versus stochastic
- Episodic versus sequential
- Static versus dynamic
- Discrete versus continuous
- Nature of Agents
- Autonomous versus Semi-Autonomous
- Reflexive, Goal-based, and Utility-based
- The importance of perception and environmental interactions
- Philosophical and ethical issues [elective]

Learning Outcomes:

1. Describe Turing test and the "Chinese Room" thought experiment. [Knowledge]
2. Differentiate between the concepts of optimal reasoning/behavior and human-like reasoning/behavior. [Knowledge]
3. Describe a given problem domain using the characteristics of the environments in which intelligent systems must function. [Evaluation]

IS/Basic Search Strategies

[4 Core-Tier2 hours]

(Cross-reference AL/Basic Analysis, AL/Algorithmic Strategies, AL/Fundamental Data Structures and Algorithms)

Topics:

- Problem spaces (states, goals and operators), problem solving by search
- Factored representation (factoring state into variables)
- Uninformed search (breadth-first, depth-first, depth-first with iterative deepening)
- Heuristics and informed search (hill-climbing, generic best-first, A*)
- Space and time efficiency of search
- Two-player games (Introduction to minimax search)
- Constraint satisfaction (backtracking and local search methods)

Learning Outcomes:

1. Formulate an efficient problem space for a problem expressed in natural language (e.g., English) in terms of initial and goal states, and operators. [Application]
2. Describe the role of heuristics and describe the trade-offs among completeness, optimality, time complexity, and space complexity. [Knowledge]
3. Describe the problem of combinatorial explosion of search space and its consequences. [Knowledge]
4. Select and implement an appropriate uninformed search algorithm for a problem, and characterize its time and space complexities. [Evaluation, Application]
5. Select and implement an appropriate informed search algorithm for a problem by designing the necessary heuristic evaluation function. [Evaluation, Application]
6. Evaluate whether a heuristic for a given problem is admissible/can guarantee optimal solution. [Evaluation]
7. Formulate a problem specified in natural language (e.g., English) as a constraint-satisfaction problem and implement it using a chronological backtracking algorithm or stochastic local search. [Application]
8. Compare and contrast basic search issues with game playing issues [Knowledge]

IS/Basic Knowledge Representation and Reasoning

[3 Core-Tier2 hours]

Topics:

- Review of propositional and predicate logic (cross-reference DS/Basic Logic)
- Resolution and theorem proving, unification and lifting (propositional logic only)
- Forward chaining, backward chaining
- Review of probabilistic reasoning, Bayes theorem (cross-reference with DS/Discrete Probability)

Learning Outcomes:

1. Translate a natural language (e.g., English) sentence into predicate logic statement. [Application]
2. Convert a quantified logic statement into clause form. [Application]
3. Apply resolution to a set of logic statements to answer a query. [Application]
4. Apply Bayes theorem to determine conditional probabilities in a problem. [Application]

IS/Basic Machine Learning

[2 Core-Tier2 hours]

Topics:

- Definition and examples of machine learning for classification
- Inductive learning
- Simple statistical-based learning such as Naive Bayesian Classifier, Decision trees
- Define overfitting problem
- Measuring classifier accuracy

Learning Outcomes:

1. Identify examples of classification tasks, including the available input features and output to be predicted. [Knowledge]
2. Explain the difference between inductive and deductive learning. [Knowledge]
3. Apply the simple statistical learning algorithm such as Naive Bayesian Classifier to a classification task and measure the classifier's accuracy. [Application]

IS/Advanced Search

[Elective]

Topics:

- Constructing search trees, dynamic search space, combinatorial explosion of search space
- Stochastic search
- Simulated annealing
- Genetic algorithms
- A* search, Beam search
- Minimax Search, Alpha-beta pruning
- Expectimax search (MDP-solving) and chance nodes

Learning Outcomes:

1. Design and implement a genetic algorithm solution to a problem. [Application]
2. Design and implement a simulated annealing schedule to avoid local minima in a problem. [Application]
3. Design and implement A*/beam search to solve a problem. [Application]
4. Apply minimax search with alpha-beta pruning to prune search space in a two-player game. [Application]
5. Compare and contrast genetic algorithms with classic search techniques. [Evaluation]
6. Compare and contrast various heuristic searches vis-a-vis applicability to a given problem. [Evaluation]

IS/Advanced Representation and Reasoning

[Elective]

Topics:

- Knowledge representation issues
- Description logics
- Ontology engineering
- Non-monotonic reasoning
- Non-classical logics
- Default reasoning
- Belief revision
- Preference logics
- Integration of knowledge sources
- Aggregation of conflicting belief
- Reasoning about action and change
- Situation calculus
- Event calculus
- Ramification problems
- Temporal and spatial reasoning
- Rule-based Expert Systems
- Model-based and Case-based reasoning
- Planning:
- Partial and totally ordered planning
- Plan graphs
- Hierarchical planning
- Planning and execution including conditional planning and continuous planning
- Mobile agent/Multi-agent planning

Learning Outcomes:

1. Compare and contrast the most common models used for structured knowledge representation, highlighting their strengths and weaknesses. [Evaluation]
2. Identify the components of non-monotonic reasoning and its usefulness as a representational mechanisms for belief systems. [Knowledge]
3. Compare and contrast the basic techniques for representing uncertainty. [Knowledge, Evaluation]
4. Compare and contrast the basic techniques for qualitative representation. [Knowledge, Evaluation]
5. Apply situation and event calculus to problems of action and change. [Application]
6. Explain the distinction between temporal and spatial reasoning, and how they interrelate. [Knowledge, Evaluation]
7. Explain the difference between rule-based, case-based and model-based reasoning techniques. [Knowledge, Evaluation]
8. Define the concept of a planning system and how they differ from classical search techniques. [Knowledge, Evaluation]
9. Describe the differences between planning as search, operator-based planning, and propositional planning, providing examples of domains where each is most applicable. [Knowledge, Evaluation]
10. Explain the distinction between monotonic and non-monotonic inference. [Knowledge]

IS/Reasoning Under Uncertainty

[Elective]

Topics:

- Review of basic probability (cross-reference DS/Discrete Probability)
- Unconditional/prior probabilities
- Conditional/posterior probabilities
- Random variables and probability distributions
- Axioms of probability
- Probabilistic inference
- Bayes' Rule
- Conditional Independence
- Knowledge representations
- Bayesian Networks
- Exact inference and its complexity
- Randomized sampling (Monte Carlo) methods (e.g. Gibbs sampling)
- Markov Networks
- Relational probability models
- Hidden Markov Models
- Decision Theory
- Preferences and utility functions
- Maximizing expected utility

Learning Outcomes:

1. Apply Bayes' rule to determine the probability of a hypothesis given evidence. [Application]
2. Explain how conditional independence assertions allow for greater efficiency of probabilistic systems. [Evaluation]
3. Identify examples of knowledge representations for reasoning under uncertainty. [Knowledge]
4. State the complexity of exact inference. Identify methods for approximate inference. [Knowledge]
5. Design and implement at least one knowledge representation for reasoning under uncertainty. [Application]
6. Describe the complexities of temporal probabilistic reasoning. [Knowledge]
7. Explain the complexities of temporal probabilistic reasoning. [Evaluation]

8. Design and implement an HMM as one example of a temporal probabilistic system. [Application]
9. Describe the relationship between preferences and utility functions. [Knowledge]
10. Explain how utility functions and probabilistic reasoning can be combined to make rational decisions. [Evaluation]

IS/Agents

[Elective]

(Cross-reference HC/Collaboration and Communication)

Topics:

- Definitions of agents
- Agent architectures
- Simple reactive agents
- Reactive planners
- Layered architectures
- Cognitive architectures
- Integrated architecture
- Example architectures and applications
- Agent theory
- Rationality, Game Theory
- Commitments
- Intentions
- Decision-theoretic agents
- Markov decision processes (MDP)
- Software agents, personal assistants, and information access
- Collaborative agents
- Information-gathering agents
- Believable agents (synthetic characters, modeling emotions in agents)
- Learning agents
- Multi-agent systems
- Collaborating agents
- Agent teams
- Competitive agents
- Game theory
- Voting
- Auctions
- Swarm systems and biologically inspired models

Learning Outcomes:

1. List the defining characteristics of an intelligent agent. [Knowledge]
2. Characterize and contrast the standard agent architectures. [Evaluation]
3. Describe the applications of agent theory to domains such as software agents, personal assistants, and believable agents. [Knowledge]
4. Describe the primary paradigms used by learning agents. [Knowledge]
5. Demonstrate using appropriate examples how multi-agent systems support agent interaction. [Application]

IS/Natural Language Processing

[Elective]

(Cross-reference HC/Design for non-mouse Interfaces)

Topics:

- Deterministic and stochastic grammars
- Parsing algorithms
- CFGs and chart parsers (e.g. CYK)
- Probabilistic CFGs and weighted CYK
- Representing meaning / Semantics
- Logic-based knowledge representations
- Semantic roles
- Temporal representations
- Verbs and event types
- Beliefs, desires, and intentions
- Ambiguity
- Long-distance dependencies
- Corpus-based methods
- N-grams and HMMs
- Smoothing and backoff
- Perplexity
- Zipf's law
- Examples of use: POS tagging and morphology
- Information retrieval (Cross-reference IM/Information Storage and Retrieval)
- Vector space model
- TF & IDF
- Precision and recall
- Information extraction
- Language translation
- Transfer-based models
- Statistical, phrase-based models
- Text classification, categorization
- Bag of words model

Learning Outcomes:

1. Define and contrast deterministic and stochastic grammars, providing examples to show the adequacy of each. [Evaluation]
2. Simulate, apply, or implement classic and stochastic algorithms for parsing natural language. [Application]
3. Identify the challenges of representing meaning. [Knowledge]
4. List the advantages of using standard corpora. Identify examples of current corpora for a variety of NLP tasks. [Knowledge]
5. Identify techniques for information retrieval, language translation, and text classification. [Knowledge]

IS/Advanced Machine Learning

[Elective]

Topics:

- Definition and examples of broad variety of machine learning tasks
- General statistical-based learning, parameter estimation (maximum likelihood)
- Inductive logic programming (ILP)
- Supervised learning
- Learning decision trees
- Learning neural networks
- Support vector machines (SVMs)
- Ensembles
- Nearest-neighbor algorithms
- Unsupervised Learning and clustering
- EM
- K-means
- Self-organizing maps
- Semi-supervised learning
- Learning graphical models (Cross-reference IS/Reasoning under Uncertainty)
- Performance evaluation (such as cross-validation, area under ROC curve)
- Learning theory
- The problem of overfitting, the curse of dimensionality
- Reinforcement learning
- Exploration vs. exploitation trade-off
- Markov decision processes
- Value and policy iteration
- Application of Machine Learning algorithms to Data Mining (Cross-reference IM/Data Mining)

Learning Outcomes:

1. Explain the differences among the three main styles of learning: supervised, reinforcement, and unsupervised. [Knowledge]
2. Implement simple algorithms for supervised learning, reinforcement learning, and unsupervised learning. [Application]
3. Determine which of the three learning styles is appropriate to a particular problem domain. [Application]
4. Compare and contrast each of the following techniques, providing examples of when each strategy is superior: decision trees, neural networks, and belief networks. [Evaluation]
5. Evaluate the performance of a simple learning system on a real-world dataset. [Evaluation]
6. Characterize the state of the art in learning theory, including its achievements and its shortcomings. [Knowledge]
7. Explain the problem of overfitting, along with techniques for detecting and managing the problem. [Application]

IS/Robotics

[Elective]

Topics:

- Overview: problems and progress
- State-of-the-art robot systems, including their sensors and an overview of their sensor processing
- Robot control architectures, e.g., deliberative vs. reactive control and Braitenberg vehicles
- World modeling and world models
- Inherent uncertainty in sensing and in control
- Configuration space and environmental maps
- Interpreting uncertain sensor data
- Localizing and mapping
- Navigation and control
- Motion planning
- Multiple-robot coordination

Learning Outcomes:

1. List capabilities and limitations of today's state-of-the-art robot systems, including their sensors and the crucial sensor processing that informs those systems. [Knowledge]
2. Integrate sensors, actuators, and software into a robot designed to undertake some task. [Application]
3. Program a robot to accomplish simple tasks using deliberative, reactive, and/or hybrid control architectures. [Application]
4. Implement fundamental motion planning algorithms within a robot configuration space. [Application]
5. Characterize the uncertainties associated with common robot sensors and actuators; **articulate** strategies for mitigating these uncertainties. [Knowledge]
6. List the differences among robots' representations of their external environment, including their strengths and shortcomings. [Knowledge]
7. Compare and contrast at least three strategies for robot navigation within known and/or unknown environments, including their strengths and shortcomings. [Evaluation]
8. Describe at least one approach for coordinating the actions and sensing of several robots to accomplish a single task. [Knowledge]

IS/Perception and Computer Vision

[Elective]

Topics:

- Computer vision
- Image acquisition, representation, and properties
- Image pre-processing via linear and nonlinear filtering
- Foreground/background segmentation
- Shape representation and object recognition
- Image inference based on prior models, i.e., image understanding
- Motion analysis
- Other modes of sensing
- Audio and speech recognition
- Sensory transformations
- Modularity in recognition
- Raw signals, acquisition issues, and sources of noise

- Task-independent features, e.g., image edges or phonetic frames
- Percepts as collections of features, e.g., edge-based contours or word-level hypotheses
- Task-dependent features and percepts: the importance and use of prior models
- Approaches to pattern recognition [overlapping with machine learning]
- Classification algorithms and measures of classification quality
- Statistical techniques

Learning Outcomes:

1. Summarize the importance of image and object recognition in AI and indicate several significant applications of this technology. [Knowledge]
2. List at least three image-segmentation approaches, such as thresholding, edge-based and region-based algorithms, along with their defining characteristics, strengths, and weaknesses. [Knowledge]
3. Implement 2d object recognition based on contour- and/or region-based shape representations. [Application]
4. Distinguish the goals of sound-recognition, speech-recognition, and speaker-recognition and identify how the raw audio signal will be handled differently in each of these cases. [Knowledge]
5. Provide at least two examples of a transformation of a data source from one sensory domain to another, e.g., tactile data interpreted as single-band 2d images. [Knowledge]
6. Implement a feature-extraction algorithm on real data, e.g., an edge or corner detector for images or vectors of Fourier coefficients describing a short slice of audio signal. [Application]
7. Implement an algorithm combining features into higher-level percepts, e.g., a contour or polygon from visual primitives or phoneme hypotheses from an audio signal. [Application]
8. Implement a classification algorithm that segments input percepts into output categories and quantitatively evaluates the resulting classification. [Application]
9. Evaluate the performance of the underlying feature-extraction, relative to at least one alternative possible approach (whether implemented or not) in its contribution to the classification task (8), above. [Evaluation]
10. Describe at least three classification approaches, their prerequisites for applicability, their strengths, and their shortcomings. [Knowledge]

Networking and Communication (NC)

The Internet and computer networks are now ubiquitous and a growing number of computing activities strongly depend on the correct operation of the underlying network. Networks, both fixed and mobile, are a key part of today's and tomorrow's computing environment. Many computing applications that are used today would not be possible without networks. This dependency on the underlying network is likely to increase in the future.

The high-level learning objective of this module can be summarized as follows:

- Thinking in a networked world. The world is more and more interconnected and the use of networks will continue to increase. Students must understand how the network behaves and the key principles behind the organization and the operation of the computer networks.
- Continued study. The networking domain is rapidly evolving and a first networking course should be a starting point to other more advanced courses on network design, network management, sensor networks, etc.
- Principles and practice interact. Networking is real and many of the design choices that involve networks also depend on practical constraints. Students should be exposed to these practical constraints by experimenting with networking, using tools, and writing networked software.

There are different ways of organizing a networking course. Some educators prefer a top-down approach, i.e. the course starts from the applications and then explains reliable delivery, routing and forwarding, etc. Other educators prefer a bottom-up approach where the students start with the lower layers and build their understanding of the network, transport and application layers later.

NC. Networking and Communication (3 Core-Tier1 hours, 7 Core-Tier2 hours)

	Core-Tier1 hours	Core-Tier2 hours	Includes Electives
NC/Introduction	**1.5**		**N**
NC/Networked Applications	**1.5**		**N**
NC/Reliable Data Delivery		**2**	**N**
NC/Routing And Forwarding		**1.5**	**N**
NC/Local Area Networks		**1.5**	**N**
NC/Resource Allocation		**1**	**N**
NC/Mobility		**1**	**N**

NC/Introduction

[1.5 Core-Tier1 hours]

Topics:

[Core-Tier1]

- Organization of the Internet (Internet Service Providers, Content Providers, etc.)
- Switching techniques (Circuit, packet, etc.)
- Physical pieces of a network (hosts, routers, switches, ISPs, wireless, LAN, access point, firewalls, etc.)
- Layering principles (encapsulation, multiplexing)
- Roles of the different layers (application, transport, network, datalink, physical)

Learning Outcomes:

1. Articulate the organization of the Internet [Knowledge]
2. List and define the appropriate network terminology [Knowledge]
3. Describe the layered structure of a typical networked architecture [Knowledge]
4. Identify the different levels of complexity in a network (edges, core, etc.) [Knowledge]

NC/Networked Applications

[1.5 Core-Tier1 hours]

Topics:

[Core-Tier1]

- Naming and address schemes (DNS, IP addresses, Uniform Resource Identifiers, etc.)
- Distributed applications (client/server, peer-to-peer, cloud, etc.)
- HTTP as an application layer protocol
- Multiplexing with TCP and UDP
- Socket APIs

Learning Outcomes:

1. List the differences and the relations between names and addresses in a network [Knowledge]
2. Define the principles behind DNS and HTTP [Knowledge]
3. Be able to implement a simple client-server socket-based application [Knowledge]

NC/Reliable Data Delivery

[2 Core-Tier2 hours]

This Knowledge Unit is related to SF-Systems Fundamentals. Cross-reference SF/State-State Transition and SF/Reliability through Redundancy.

Topics:

[Core-Tier2]

- Error control (retransmission techniques, timers)
- Flow control (acknowledgements, sliding window)
- Performance issues (pipelining)
- TCP

Learning Outcomes:

1. Describe the operation of reliable delivery protocols [Knowledge]
2. List the factors that affect the performance of reliable delivery protocols [Knowledge]
3. Design and implement a simple reliable protocol [Application]

NC/Routing And Forwarding

[1.5 Core-Tier2 hours]

Topics:

[Core-Tier2]

- Routing versus forwarding
- Static routing
- Internet Protocol (IP)
- Scalability issues (hierarchical addressing)

Learning Outcomes:

1. Describe the organization of the network layer [Knowledge]
2. Describe how packets are forwarded in an IP networks [Knowledge]
3. List the scalability benefits of hierarchical addressing [Knowledge]

NC/Local Area Networks

[1.5 Core-Tier2 hours]

Topics:

[Core-Tier2]

- Multiple Access
- Local Area Networks
- Ethernet
- Switching

Learning Outcomes:

1. List the major steps in solving the multiple access problem [Application]
2. Describe how frames are forwarded in an Ethernet network [Knowledge]
3. Identify the differences between IP and Ethernet [Knowledge]
4. Describe the interrelations between IP and Ethernet [Application]

NC/Resource Allocation

[1 Core-Tier2 hours]

Topics:

[Core-Tier2]

- Need for resource allocation
- Fixed allocation (TDM, FDM, WDM) versus dynamic allocation
- End-to-end versus network assisted approaches
- Fairness
- Principles of congestion control

Learning Outcomes:

1. Describe how resources can be allocated in a network [Knowledge]
2. Describe the congestion problem in a large network [Knowledge]
3. Compare and contrast the fixed and dynamic allocation techniques [Evaluation]

NC/Mobility

[1 Core-Tier2 hours]

Topics:

[Core-Tier2]

- Principles of cellular networks
- 802.11 networks
- Issues in supporting mobile nodes (home agents)

Learning Outcomes:

1. Describe the organization of a wireless network [Knowledge]
2. Describe how wireless networks support mobile users [Knowledge]

Operating Systems (OS)

An operating system defines an abstraction of hardware and manages resource sharing among the computer's users. The topics in this area explain the most basic knowledge of operating systems in the sense of interfacing an operating system to networks, teaching the difference between the kernel and user modes, and developing key approaches to operating system design and implementation. This knowledge area is structured to be complementary to Systems Fundamentals, Networks, Information Assurance, and the Parallel and Distributed Computing knowledge areas. The Systems Fundamentals and Information Assurance knowledge areas are the new ones to include contemporary issues. For example, the Systems Fundamentals includes topics such as performance, virtualization and isolation, and resource allocation and scheduling; Parallel and Distributed Systems knowledge area includes parallelism fundamentals; and Information Assurance includes forensics and security issues in depth. Many courses in Operating Systems will draw material from across these Knowledge Areas.

OS. Operating Systems (4 Core-Tier1 hours; 11 Core Tier2 hours)

	Core-Tier1 hours	Core-Tier2 hours	Includes Electives
OS/Overview of Operating Systems	2		N
OS/Operating System Principles	2		N
OS/Concurrency		3	N
OS/Scheduling and Dispatch		3	N
OS/Memory Management		3	N
OS/Security and Protection		2	N
OS/Virtual Machines			Y
OS/Device Management			Y
OS/File Systems			Y
OS/Real Time and Embedded Systems			Y
OS/Fault Tolerance			Y
OS/System Performance Evaluation			Y

OS/Overview of Operating Systems

[2 Core-Tier1 hours]

Topics:

- Role and purpose of the operating system
- Functionality of a typical operating system
- Mechanisms to support client-server models, hand-held devices
- Design issues (efficiency, robustness, flexibility, portability, security, compatibility)
- Influences of security, networking, multimedia, windows

Learning Objectives:

1. Explain the objectives and functions of modern operating systems [Knowledge].
2. Analyze the tradeoffs inherent in operating system design [Application].
3. Describe the functions of a contemporary operating system with respect to convenience, efficiency, and the ability to evolve [Knowledge].
4. Discuss networked, client-server, distributed operating systems and how they differ from single user operating systems [Knowledge].
5. Identify potential threats to operating systems and the security features design to guard against them [Knowledge].

OS/Operating System Principles

[2 core-T1 hours]

Topics:

- Structuring methods (monolithic, layered, modular, micro-kernel models)
- Abstractions, processes, and resources
- Concepts of application program interfaces (APIs)
- Application needs and the evolution of hardware/software techniques
- Device organization
- Interrupts: methods and implementations
- Concept of user/system state and protection, transition to kernel mode

Learning Objectives:

1. Explain the concept of a logical layer [Knowledge].
2. Explain the benefits of building abstract layers in hierarchical fashion [Knowledge].
3. Defend the need for APIs and middleware [Evaluation].
4. Describe how computing resources are used by application software and managed by system software [Knowledge].
5. Contrast kernel and user mode in an operating system [Application].
6. Discuss the advantages and disadvantages of using interrupt processing [Knowledge].
7. Explain the use of a device list and driver I/O queue [Knowledge].

OS/Concurrency

[3 Core-Tier2 hours]

Topics:

- States and state diagrams (cross reference SF/State-State Transition-State Machines)
- Structures (ready list, process control blocks, and so forth)
- Dispatching and context switching
- The role of interrupts
- Managing atomic access to OS objects
- Implementing synchronization primitives
- Multiprocessor issues (spin-locks, reentrancy) (cross reference SF/Parallelism)

Learning Objectives:

1. Describe the need for concurrency within the framework of an operating system [Knowledge].
2. Demonstrate the potential run-time problems arising from the concurrent operation of many separate tasks [Application].
3. Summarize the range of mechanisms that can be employed at the operating system level to realize concurrent systems and describe the benefits of each [Knowledge].
4. Explain the different states that a task may pass through and the data structures needed to support the management of many tasks [Knowledge].
5. Summarize techniques for achieving synchronization in an operating system (e.g., describe how to implement a semaphore using OS primitives) [Knowledge].
6. Describe reasons for using interrupts, dispatching, and context switching to support concurrency in an operating system [Knowledge].
7. Create state and transition diagrams for simple problem domains [Application].

OS/Scheduling and Dispatch

[3 Core-Tier2 hours]

Topics:

- Preemptive and nonpreemptive scheduling (cross reference SF/Resource Allocation and Scheduling, PD/Parallel Performance)
- Schedulers and policies (cross reference SF/Resource Allocation and Scheduling, PD/Parallel Performance)
- Processes and threads (cross reference SF/computational paradigms)
- Deadlines and real-time issues

Learning Objectives:

1. Compare and contrast the common algorithms used for both preemptive and non-preemptive scheduling of tasks in operating systems, such as priority, performance comparison, and fair-share schemes [Application].
2. Describe relationships between scheduling algorithms and application domains [Knowledge].
3. Discuss the types of processor scheduling such as short-term, medium-term, long-term, and I/O [Knowledge].
4. Describe the difference between processes and threads [Application].
5. Compare and contrast static and dynamic approaches to real-time scheduling [Application].
6. Discuss the need for preemption and deadline scheduling [Knowledge].
7. Identify ways that the logic embodied in scheduling algorithms are applicable to other domains, such as disk I/O, network scheduling, project scheduling, and problems beyond computing [Application].

OS/Memory Management

[3 Core-Tier2 hours]

Topics:

- Review of physical memory and memory management hardware
- Working sets and thrashing
- Caching

Learning Objectives:

1. Explain memory hierarchy and cost-performance trade-offs [Knowledge].
2. Summarize the principles of virtual memory as applied to caching and paging [Knowledge].
3. Evaluate the trade-offs in terms of memory size (main memory, cache memory, auxiliary memory) and processor speed [Evaluation].
4. Defend the different ways of allocating memory to tasks, citing the relative merits of each [Evaluation].
5. Describe the reason for and use of cache memory (performance and proximity, different dimension of how caches complicate isolation and VM abstraction) [Knowledge].
6. Discuss the concept of thrashing, both in terms of the reasons it occurs and the techniques used to recognize and manage the problem [Knowledge].

OS/Security and Protection

[2 Core-Tier2 hours]

Topics:

- Overview of system security
- Policy/mechanism separation
- Security methods and devices
- Protection, access control, and authentication
- Backups

Learning Objectives:

1. Defend the need for protection and security in an OS (cross reference IAS/Security Architecture and Systems Administration/Investigating Operating Systems Security for various systems) [Evaluation].
2. Summarize the features and limitations of an operating system used to provide protection and security (cross reference IAS/Security Architecture and Systems Administration) [Knowledge].
3. Explain the mechanisms available in an OS to control access to resources (cross reference IAS/Security Architecture and Systems Administration/Access Control/Configuring systems to operate securely as an IT system) [Knowledge].
4. Carry out simple system administration tasks according to a security policy, for example creating accounts, setting permissions, applying patches, and arranging for regular backups (cross reference IAS/Security Architecture and Systems Administration) [Application].

OS/Virtual Machines

[Elective]

Topics:

- Types of virtualization (Hardware/Software, OS, Server, Service, Network, etc.)
- Paging and virtual memory
- Virtual file systems
- Virtual file
- Hypervisors
- Portable virtualization; emulation vs. isolation
- Cost of virtualization

Learning Objectives:

1. Explain the concept of virtual memory and how it is realized in hardware and software [Knowledge].
2. Differentiate emulation and isolation [Knowledge].
3. Evaluate virtualization trade-offs [Evaluation].
4. Discuss hypervisors and the need for them in conjunction with different types of hypervisors [Application].

OS/Device Management

[Elective]

Topics:

- Characteristics of serial and parallel devices
- Abstracting device differences
- Buffering strategies
- Direct memory access
- Recovery from failures

Learning Objectives:

1. Explain the key difference between serial and parallel devices and identify the conditions in which each is appropriate [Knowledge].
2. Identify the relationship between the physical hardware and the virtual devices maintained by the operating system [Application].
3. Explain buffering and describe strategies for implementing it [Knowledge].
4. Differentiate the mechanisms used in interfacing a range of devices (including hand-held devices, networks, multimedia) to a computer and explain the implications of these for the design of an operating system [Application].
5. Describe the advantages and disadvantages of direct memory access and discuss the circumstances in which its use is warranted [Application].
6. Identify the requirements for failure recovery [Knowledge].
7. Implement a simple device driver for a range of possible devices [Application].

OS/File Systems

[Elective]

Topics:

- Files: data, metadata, operations, organization, buffering, sequential, nonsequential
- Directories: contents and structure
- File systems: partitioning, mount/unmount, virtual file systems
- Standard implementation techniques
- Memory-mapped files
- Special-purpose file systems
- Naming, searching, access, backups
- Journaling and log-structured file systems

Learning Objectives:

1. Summarize the full range of considerations in the design of file systems [Knowledge].
2. Compare and contrast different approaches to file organization, recognizing the strengths and weaknesses of each [Application].
3. Summarize how hardware developments have led to changes in the priorities for the design and the management of file systems [Knowledge].
4. Summarize the use of journaling and how log-structured file systems enhance fault tolerance [Knowledge].

OS/Real Time and Embedded Systems

[Elective]

Topics:

- Process and task scheduling
- Memory/disk management requirements in a real-time environment
- Failures, risks, and recovery
- Special concerns in real-time systems

Learning Objectives:

1. Describe what makes a system a real-time system [Knowledge].
2. Explain the presence of and describe the characteristics of latency in real-time systems [Knowledge].
3. Summarize special concerns that real-time systems present and how these concerns are addressed [Knowledge].

OS/Fault Tolerance

[Elective]

Topics:

- Fundamental concepts: reliable and available systems (cross reference SF/Reliability through Redundancy)
- Spatial and temporal redundancy (cross reference SF/Reliability through Redundancy)
- Methods used to implement fault tolerance
- Examples of OS mechanisms for detection, recovery, restart to implement fault tolerance, use of these techniques for the OS's own services

Learning Objectives:

1. Explain the relevance of the terms fault tolerance, reliability, and availability [Knowledge].
2. Outline the range of methods for implementing fault tolerance in an operating system [Knowledge].
3. Explain how an operating system can continue functioning after a fault occurs [Knowledge].

OS/System Performance Evaluation

[Elective]

Topics:

- Why system performance needs to be evaluated (cross reference SF/Performance/Figures of performance merit)
- What is to be evaluated (cross reference SF/Performance/Figures of performance merit)
- Policies for caching, paging, scheduling, memory management, security, and so forth
- Evaluation models: deterministic, analytic, simulation, or implementation-specific
- How to collect evaluation data (profiling and tracing mechanisms)

Learning Objectives:

1. Describe the performance measurements used to determine how a system performs [Knowledge].
2. Explain the main evaluation models used to evaluate a system [Knowledge].

Platform-Based Development (PBD)

Platform-based development is concerned with the design and development of software applications that reside on specific software platforms. In contrast to general purpose programming, platform-based development takes into account platform-specific constraints. For instance web programming, multimedia development, mobile computing, app development, and robotics are examples of relevant platforms which provide specific services/APIs/hardware which constrain development. Such platforms are characterized by the use of specialized APIs, distinct delivery/update mechanisms, and being abstracted away from the machine level. Platform-based development may be applied over a wide breadth of ecosystems.

While we recognize that some platforms (e.g., web development) are prominent, we are also cognizant of the fact that no particular platform should be specified as a requirement in the CS2013 curricular guidelines. Consequently, this Knowledge Area highlights many of the platforms which have become popular, without including any such platform in the core curriculum. We note that the general skill of developing with respect to an API or a constrained environment is covered in other Knowledge Areas, such as SDF-Software Development Fundamentals. Platform-based development further emphasizes such general skills within the context of particular platforms.

PBD. Platform-Based Development (Elective)

	Core-Tier1 hours	**Core-Tier2 hours**	**Includes Electives**
PBD/Introduction			**Y**
PBD/Web Platforms			**Y**
PBD/Mobile Platforms			**Y**
PBD/Industrial Platforms			**Y**
PBD/Game Platforms			**Y**

PBD/Introduction

[Elective]

This unit describes the fundamental differences that Platform-Based Development has over traditional software development.

Topics:

- Overview of platforms (Web, Mobile, Game, Industrial etc)
- Programming via platform-specific APIs
- Overview of Platform Languages (Objective C, HTML5, etc)
- Programming under platform constraints

Learning Outcomes:

1. Describe how platform-based development differs from general purpose programming [Knowledge]
2. List characteristics of platform languages [Knowledge]
3. Write and execute a simple platform-based program [Application]
4. List the advantages and disadvantages of programming with platform constraints [Knowledge]

PBD/Web Platforms

[Elective]

Topics:

- Web programming languages (HTML5, Java Script, PHP, CSS, etc.)
- Web platform constraints
- Software as a Service (SaaS)

Learning Outcomes:

1. Design and Implement a simple web application [Application]
2. Describe the constraints that the web puts on developers [Knowledge]
3. Compare and contrast web programming with general purpose programming [Evaluation]
4. Describe the differences between Software-as-a-Service and traditional software products [Knowledge]

PBD/Mobile Platforms

[Elective]

Topics:

- Mobile Programming Languages (Objective C, Java Script, Java, etc.)
- Challenges with mobility and wireless communication
- Location-aware applications
- Performance / power tradeoffs
- Mobile platform constraints
- Emerging Technologies

Learning Outcomes:

1. Design and implement a mobile application for a given mobile platform. [Application]
2. Discuss the constraints that mobile platforms put on developers [Knowledge]
3. Discuss the performance vs. power tradeoff [Knowledge]
4. Compare and Contrast mobile programming with general purpose programming [Evaluation]

PBD/Industrial Platforms

[Elective]

This knowledge unit is related to IS/Robotics.

Topics:

- Types of Industrial Platforms (Mathematic, Robotics, Industrial Controls, etc.)
- Robotic Software and its Architecture
- Domain Specific Languages
- Industrial Platform Constraints

Learning Outcomes:

1. Design and implement an industrial application on a given platform (Lego Mindstorms, Matlab, etc.) [Application]
2. Compare and contrast domain specific languages with general purpose programming languages. [Evaluate]
3. Discuss the constraints that a given industrial platforms impose on developers [Knowledge]

PBD/Game Platforms

[Elective]

Topics:

- Types of Game Platforms (XBox, Wii, PlayStation, etc)
- Game Platform Languages (C++, Java, Lua, Python, etc)
- Game Platform Constraints

Learning Outcomes:

1. Design and Implement a simple application on a game platform. [Application]
2. Describe the constraints that game platforms impose on developers. [Knowledge]
3. Compare and contrast game programming with general purpose programming [Evaluation]

Parallel and Distributed Computing (PD)

The past decade has brought explosive growth in multiprocessor computing, including multi-core processors and distributed data centers. As a result, parallel and distributed computing has moved from a largely elective topic to become more of a core component of undergraduate computing curricula. Both parallel and distributed computing entail the logically simultaneous execution of multiple processes, whose operations have the potential to interleave in complex ways. Parallel and distributed computing builds on foundations in many areas, including an understanding of fundamental systems concepts such as concurrency and parallel execution, consistency in state/memory manipulation, and latency. Communication and coordination among processes is rooted in the message-passing and shared-memory models of computing, the system goals of concurrency and speedup, and such algorithmic concepts as atomicity, consensus, and conditional waiting. Achieving speedup in practice requires an understanding of parallel algorithms, strategies for problem decomposition, system architecture, and performance analysis and tuning. Distributed systems highlight the problems of security and fault tolerance, emphasize the maintenance of replicated state, and introduce additional issues that bridge to computer networking.

Because parallelism interacts with so many areas of computing, including at least algorithms, languages, systems, networking, and hardware, many curricula will put different parts of the knowledge area in different courses, rather than in a dedicated course. While we acknowledge that computer science is moving in this direction and may reach that point, in 2013 this process is still in flux and we feel it provides more useful guidance to curriculum designers to aggregate the fundamental parallelism topics in one place. Note, however, that the fundamentals of concurrency and mutual exclusion appear in Systems Fundamentals. Many curricula may choose to introduce parallelism and concurrency in the same course. Further, we note that the topics and learning outcomes listed below include only brief mentions of purely elective coverage. At the present time, there is too much diversity in topics that share little in common (including for example, parallel scientific computing, process calculi, and non-blocking data structures) to recommend particular topics be covered in elective courses.

Because the terminology of parallel and distributed computing varies among communities, we provide here brief descriptions of the intended senses of a few terms. This list is not exhaustive or definitive, but is provided for the sake of clarity:

- *Activity*: A computation that may proceed concurrently with others; for example a program, process, thread, or active parallel hardware component.
- *Atomicity*: Rules and properties governing whether an action is observationally indivisible; for example setting all of the bits in a word, transmitting a single packet, or completing a transaction.
- *Consensus*: Agreement among two or more activities about a given predicate; for example the value of a counter, the owner of a lock, or the termination of a thread.
- *Consistency*: Rules and properties governing agreement about the values of variables written, or messages produced, by some activities and used by others (thus possibly exhibiting a *data race*); for example, *sequential consistency*, stating that the values of all variables in a shared memory parallel program are equivalent to that of a single program performing some interleaving of the memory accesses of these activities.
- *Multicast*: A message sent to possibly many recipients, generally without any constraints about whether some recipients receive the message before others. An *event* is a multicast message sent to a designated set of *listeners* or *subscribers*.

As multi-processor computing continues to grow in the coming years, so too will the role of parallel and distributed computing in undergraduate computing curricula. In addition to the guidelines presented here, we also direct the interested reader to the document entitled "NSF/TCPP Curriculum Initiative on Parallel and Distributed Computing - Core Topics for Undergraduates", available from the website: http://www.cs.gsu.edu/~tcpp/curriculum/.

General cross-referencing note: Systems Fundamentals also contains an introduction to parallelism (SF/Computational Paradigms, SF/System Support for Parallelism, SF/Performance). The introduction to parallelism in SF complements the one here and there is no ordering constraint between them. In SF, the idea is to provide a unified view of the system support for simultaneous execution at multiple levels of abstraction (parallelism is inherent in gates, processors, operating systems, servers, etc.), whereas here the focus is on a preliminary understanding of parallelism as a computing primitive and the complications that arise in parallel and concurrent programming. Given these different perspectives, the hours assigned to each are not redundant: the layered systems view and the high-level computing concepts are accounted for separately in terms of the core hours.

PD. Parallel and Distributed Computing (5 Core-Tier1 hours, 9 Core-Tier2 hours)

	Core-Tier1 hours	Core-Tier2 hours	Includes Electives
PD/Parallelism Fundamentals	2		N
PD/Parallel Decomposition	1	3	N
PD/Communication and Coordination	1	3	Y
PD/Parallel Algorithms, Analysis, and Programming		3	Y
PD/Parallel Architecture	1	1	Y
PD/Parallel Performance			Y
PD/Distributed Systems			Y
PD/Formal Models and Semantics			Y

PD/Parallelism Fundamentals

[2 Core-Tier1 hours]

Build upon students' familiarity with the notion of basic parallel execution--a concept addressed in Systems Fundamentals--to delve into the complicating issues that stem from this notion, such as race conditions and liveness.

(Cross-reference SF/Computational Paradigms and SF/System Support for Parallelism)

Topics:

[Core-Tier1]

- Multiple simultaneous computations
- Goals of parallelism (e.g., throughput) versus concurrency (e.g., controlling access to shared resources)
- Programming constructs for creating parallelism, communicating, and coordinating
- Programming errors not found in sequential programming
  - Data races (simultaneous read/write or write/write of shared state)
  - Higher-level races (interleavings violating program intention)
  - Lack of liveness/progress (deadlock, starvation)

Learning outcomes:

1. Distinguish using computational resources for a faster answer from managing efficient access to a shared resource [Knowledge]
2. Distinguish multiple sufficient programming constructs for synchronization that may be inter-implementable but have complementary advantages [Knowledge]
3. Distinguish data races from higher level races [Knowledge]

PD/Parallel Decomposition

[1 Core-Tier1 hour, 3 Core-Tier2 hours]

(Cross-reference SF/System Support for Parallelism)

Topics:

[Core-Tier1]

- Need for communication and coordination/synchronization
- Independence and partitioning

[Core-Tier2]

- Basic knowledge of parallel decomposition concepts (cross-reference SF/System Support for Parallelism)
- Task-based decomposition
- Implementation strategies such as threads
- Data-parallel decomposition
- Implementation strategies such as SIMD and MapReduce
- Actors and reactive processes (e.g., request handlers)

Learning outcomes:

1. Explain why synchronization is necessary in a specific parallel program [Application]
2. Write a correct and scalable parallel algorithm [Application]
3. Parallelize an algorithm by applying task-based decomposition [Application]
4. Parallelize an algorithm by applying data-parallel decomposition [Application]

PD/Communication and Coordination

[1 Core-Tier1 hour, 3 Core-Tier2 hours]

Topics:

[Core-Tier1]

- Shared Memory
- Sequential consistency, and its role in programming language guarantees for data-race-free programs

[Core-Tier2]

- Consistency in shared memory models
- Message passing
    - Point-to-point versus multicast (or event-based) messages
    - Blocking versus non-blocking styles for sending and receiving messages
    - Message buffering (cross-reference PF/Fundamental Data Structures/Queues)
- Atomicity
    - Specifying and testing atomicity and safety requirements
    - Granularity of atomic accesses and updates, and the use of constructs such as critical sections or transactions to describe them
    - Mutual Exclusion using locks, semaphores, monitors, or related constructs
    - Potential for liveness failures and deadlock (causes, conditions, prevention)
- Composition
- Composing larger granularity atomic actions using synchronization
- Transactions, including optimistic and conservative approaches

[Elective]

- Consensus
- (Cyclic) barriers, counters, or related constructs
- Conditional actions
- Conditional waiting (e.g., using condition variables)

Learning outcomes:

1. Use mutual exclusion to avoid a given race condition [Application]
2. Give an example of an ordering of accesses among concurrent activities that is not sequentially consistent [Knowledge]
3. Give an example of a scenario in which blocking message sends can deadlock [Application]
4. Explain when and why multicast or event-based messaging can be preferable to alternatives [Knowledge]
5. Write a program that correctly terminates when all of a set of concurrent tasks have completed [Application]
6. Use a properly synchronized queue to buffer data passed among activities [Application]

7. Explain why checks for preconditions, and actions based on these checks, must share the same unit of atomicity to be effective [Knowledge]
8. Write a test program that can reveal a concurrent programming error; for example, missing an update when two activities both try to increment a variable [Application]
9. Describe at least one design technique for avoiding liveness failures in programs using multiple locks or semaphores [Knowledge]
10. Describe the relative merits of optimistic versus conservative concurrency control under different rates of contention among updates [Knowledge]
11. Give an example of a scenario in which an attempted optimistic update may never complete [Knowledge]
12. Use semaphores or condition variables to block threads until a necessary precondition holds [Application]

PD/Parallel Algorithms, Analysis, and Programming

[3 Core-Tier2 hours]

Topics:

[Core-Tier2]

- Critical paths, work and span, and the relation to Amdahl's law (cross-reference SF/Performance)
- Speed-up and scalability
- Naturally (embarassingly) parallel algorithms
- Parallel algorithmic patterns (divide-and-conquer, map and reduce, others)
- Specific algorithms (e.g., parallel MergeSort)

[Elective]

- Parallel graph algorithms (e.g., parallel shortest path, parallel spanning tree) (cross-reference AL/Algorithmic Strategies/Divide-and-conquer)
- Producer-consumer and pipelined algorithms

Learning outcomes:

1. Define "critical path", "work", and "span" [Knowledge]
2. Compute the work and span, and determine the critical path with respect to a parallel execution diagram [application]
3. Define "speed-up" and explain the notion of an algorithm's scalability in this regard [Knowledge]
4. Identify independent tasks in a program that may be parallelized [Application]
5. Characterize features of a workload that allow or prevent it from being naturally parallelized [Knowledge]
6. Implement a parallel divide-and-conquer and/or graph algorithm and empirically measure its performance relative to its sequential analog [application]
7. Decompose a problem (e.g., counting the number of occurrences of some word in a document) via map and reduce operations [Application]
8. Provide an example of a problem that fits the producer-consumer paradigm [Knowledge]
9. Give examples of problems where pipelining would be an effective means of parallelization [Knowledge]
10. Identify issues that arise in producer-consumer algorithms and mechanisms that may be used for addressing them [Knowledge]

PD/Parallel Architecture

[1 Core-Tier1 hour, 1 Core-Tier2 hour]

The topics listed here are related to knowledge units in the Architecture and Organization area (AR/Assembly Level Machine Organization and AR/Multiprocessing and Alternative Architectures). Here, we focus on parallel architecture from the standpoint of applications, whereas the Architecture and Organization area presents the topic from the hardware perspective.

[Core-Tier1]

- Multicore processors
- Shared vs. distributed memory

[Core-Tier2]

- Symmetric multiprocessing (SMP)
- SIMD, vector processing

[Elective]

- GPU, co-processing
- Flynn's taxonomy
- Instruction level support for parallel programming
- Atomic instructions such as Compare and Set
- Memory issues
- Multiprocessor caches and cache coherence
- Non-uniform memory access (NUMA)
- Topologies [Elective]
- Interconnects
- Clusters
- Resource sharing (e.g., buses and interconnects)

Learning outcomes:

1. Describe the SMP architecture and note its key features [Knowledge]
2. Characterize the kinds of tasks that are a natural match for SIMD machines [Knowledge]
3. Explain the features of each classification in Flynn's taxonomy [Knowledge]
4. Explain the differences between shared and distributed memory [Knowledge]
5. Describe the challenges in maintaining cache coherence [Knowledge]
6. Describe the key features of different distributed system topologies [Knowledge]

PD/Parallel Performance

[Elective]

Topics:

- Load balancing
- Performance measurement
- Scheduling and contention (cross-reference OS/Scheduling and Dispatch)
- Data management
    - Non-uniform communication costs due to proximity (cross-reference SF/Proximity)
    - Cache effects (e.g., false sharing)
    - Maintaining spatial locality
- Impact of composing multiple concurrent components
- Power usage and management

Learning outcomes:

1. Calculate the implications of Amdahl's law for a particular parallel algorithm [Application]
2. Describe how data distribution/layout can affect an algorithm's communication costs [Knowledge]
3. Detect and correct a load imbalance [Application]
4. Detect and correct an instance of false sharing [Application]
5. Explain the impact of scheduling on parallel performance [Knowledge]
6. Explain performance impacts of data locality [Knowledge]
7. Explain the impact and trade-off related to power usage on parallel performance [Knowledge]

PD/Distributed Systems

[Elective]

Topics:

- Faults (cross-reference OS/Fault Tolerance)
    - Network-based (including partitions) and node-based failures
    - Impact on system wide guarantees (e.g., availability)
- Distributed message sending
    - Data conversion and transmission
    - Sockets
    - Message sequencing
    - Buffering, retrying, and dropping messages
- Distributed system design tradeoffs
    - Latency versus throughput
    - Consistency, availability, partition tolerance
- Distributed service design
    - Stateful versus stateless protocols and services
    - Session (connection-based) designs
    - Reactive (IO-triggered) and multithreaded designs
- Core distributed algorithms
    - Election, discovery
- Scaling
    - Clusters, grids, meshes, and clouds

Learning outcomes:

1. Distinguish network faults from other kinds of failures [Knowledge]
2. Explain why synchronization constructs such as simple locks are not useful in the presence of distributed faults [Knowledge]
3. Give examples of problems for which consensus algorithms such as leader election are required [Application]
4. Write a program that performs any required marshalling and conversion into message units, such as packets, to communicate interesting data between two hosts [Application]
5. Measure the observed throughput and response latency across hosts in a given network [Application]
6. Explain why no distributed system can be simultaneously consistent, available, and partition tolerant [Knowledge]
7. Implement a simple server -- for example, a spell checking service [Application]
8. Explain the tradeoffs among overhead, scalability, and fault tolerance when choosing a stateful v. stateless design for a given service [Knowledge]
9. Describe the scalability challenges associated with a service growing to accommodate many clients, as well as those associated with a service only transiently having many clients [Knowledge]

PD/Formal Models and Semantics

[Elective]

Topics:

- Formal models of processes and message passing, including algebras such as Communicating Sequential Processes (CSP) and pi-calculus
- Formal models of parallel computation, including the Parallel Random Access Machine (PRAM) and alternatives such as Bulk Synchronous Parallel (BSP)
- Models of (relaxed) shared memory consistency and their relation to programming language specifications
- Algorithmic correctness criteria including linearizability
- Models of algorithmic progress, including non-blocking guarantees and fairness
- Techniques for specifying and checking correctness properties such as atomicity and freedom from data races

Learning outcomes:

1. Model a concurrent process using a formal model, such as pi-calculus [Application]
2. Explain the characteristics of a particular formal parallel model [Knowledge]
3. Formally model a shared memory system to show if it is consistent [Application
4. Use a model to show progress guarantees in a parallel algorithm [Application]
5. Use formal techniques to show that a parallel algorithm is correct with respect to a safety or liveness property [Application]
6. Decide if a specific execution is linearizable or not [Application]

Programming Languages (PL)

Programming languages are the medium through which programmers precisely describe concepts, formulate algorithms, and reason about solutions. In the course of a career, a computer scientist will work with many different languages, separately or together. Software developers must understand the programming models underlying different languages, and make informed design choices in languages supporting multiple complementary approaches. Computer scientists will often need to learn new languages and programming constructs, and must understand the principles underlying how programming language features are defined, composed, and implemented. The effective use of programming languages, and appreciation of their limitations, also requires a basic knowledge of programming language translation and static program analysis, as well as run-time components such as memory management.

PL. Programming Languages (8 Core-Tier1 hours, 20 Core-Tier2 hours)

	Core-Tier1 hours	Core-Tier2 hours	Includes Electives
PL/Object-Oriented Programming	4	6	N
PL/Functional Programming	3	4	N
PL/Event-Driven and Reactive Programming		2	N
PL/Basic Type Systems	1	4	N
PL/Program Representation		1	N
PL/Language Translation and Execution		3	N
PL/Syntax Analysis			Y
PL/Compiler Semantic Analysis			Y
PL/Code Generation			Y
PL/Runtime Systems			Y
PL/Static Analysis			Y
PL/Advanced Programming Constructs			Y
PL/Concurrency and Parallelism			Y
PL/Type Systems			Y
PL/Formal Semantics			Y
PL/Language Pragmatics			Y
PL/Logic Programming			Y

Note:

- Some topics from one or more of the first three Knowledge Units (*Object-Oriented Programming*, *Functional Programming*, *Event-Driven and Reactive Programming*) are likely to be integrated with topics in the Software Development Fundamentals Knowledge Area in a curriculum's introductory courses. Curricula will differ on which topics are integrated in this fashion and which are delayed until later courses on software development and programming languages.
- The Knowledge Units with core hours have a unified collection of learning outcomes, which appears below these Knowledge Units.

PL/Object-Oriented Programming

[4 Core-Tier1 hours, 6 Core-Tier2 hours]

Topics:

[Core-Tier1]

- Object-oriented design
  - Decomposition into objects carrying state and having behavior
  - Class-hierarchy design for modeling
- Definition of classes: fields, methods, and constructors
- Subclasses, inheritance, and overriding
- Dynamic dispatch: definition of method-call

[Core-Tier2]

- Subtyping (cross-reference PL/Type Systems)
  - Subtype polymorphism; implicit upcasts in typed languages
  - Notion of behavioral replacement
  - Relationship between subtyping and inheritance
- Object-oriented idioms for encapsulation
  - Private fields
  - Interfaces revealing only method signatures
  - Abstract base classes
- Using collection classes, iterators, and other common library components

PL/Functional Programming

[3 Core-Tier1 hours, 4 Core-Tier2 hours]

Topics:

[Core-Tier1]

- Benefits of effect-free programming
  - Data can be freely aliased or copied without introducing unintended effects from mutation
  - Function calls have no side effects, facilitating compositional reasoning
  - Variables are immutable, preventing unexpected changes to program data by other code
- Processing structured data (e.g., trees) via functions with cases for each data variant
  - Associated language constructs such as discriminated unions and pattern-matching over them
  - Compositional functions over structured data
- First-class functions (taking, returning, and storing functions)

[Core-Tier2]

- Function closures (functions using variables in the enclosing lexical environment)
  - Basic meaning and definition -- creating closures at run-time by capturing the environment
  - Canonical idioms: call-backs, arguments to iterators, reusable code via function arguments
  - Using a closure to encapsulate data in its environment
- Defining higher-order operations on aggregates, especially map, reduce/fold, and filter

PL/Event-Driven and Reactive Programming

[2 Core-Tier2 hours]

This material can stand alone or be integrated with other knowledge units on concurrency, asynchrony, and threading to allow contrasting events with threads.

Topics:

- Events and event handlers
- Canonical uses such as GUIs, mobile devices, robots, servers
- Using a reactive framework
  - Defining event handlers/listeners
  - Main event loop not under event-handler-writer's control
- Externally-generated events and program-generated events
- Separation of model, view, and controller

PL/Basic Type Systems

[1 Core-Tier1 hour, 4 Core-Tier2 hours]

The core-tier2 hours would be profitably spent both on the core-tier2 topics and on a less shallow treatment of the core-tier1 topics.

Topics:

[Core-Tier1]

- A type as a set of values together with a set of operations
  - Primitive types (e.g., numbers, Booleans)
  - Reference types
  - Compound types built from other types (e.g., records, unions, arrays, lists, functions)
- Association of types to variables, arguments, results, and fields
- Type safety and errors caused by using values inconsistently with their intended types
- Goals and limitations of static typing
  - Eliminating some classes of errors without running the program
  - Inherent conservative approximation of static analysis due to undecidability

[Core-Tier2]

- Generic types (parametric polymorphism)
  - Definition
  - Use for generic libraries such as collections
  - Comparison with ad hoc polymorphism (overloading) and subtype polymorphism
- Complementary benefits of static and dynamic typing
  - Errors early vs. errors late/avoided
  - Enforce invariants during code maintenance vs. postpone typing decisions while prototyping
  - Avoid misuse of code vs. allow more code reuse
  - Detect incomplete programs vs. allow incomplete programs to run

PL/Program Representation

[1 Core-Tier2 hour]

Topics:

- Programs that take (other) programs as input such as interpreters, compilers, type-checkers, documentation generators, etc.
- Abstract syntax trees; contrast with concrete syntax
- Data structures to represent code for execution, translation, or transmission

PL/Language Translation and Execution

[3 Core-Tier2 hours]

Topics:

- Interpretation vs. compilation to native code vs. compilation to portable intermediate representation
- Language translation pipeline: parsing, optional type-checking, translation, linking, execution
    - Execution as native code or within a virtual machine
    - Alternatives like dynamic loading and dynamic code generation
- Run-time representation of core language constructs such as objects (method tables) and first-class functions (closures)
- Run-time layout of memory: call-stack, heap, static data
    - Implementing loops, recursion, and tail calls
- Automated vs. manual memory management; garbage collection as an automatic technique using the notion of reachability

Learning outcomes for all PL Knowledge Units with Core Topics:

1. Compare and contrast (1) the procedural/functional approach—defining a function for each operation with the function body providing a case for each data variant—and (2) the object-oriented approach—defining a class for each data variant with the class definition providing a method for each operation. Understand both as defining a matrix of operations and variants. [Evaluation]
2. Use subclassing to design simple class hierarchies that allow code to be reused for distinct subclasses. [Application]
3. Use multiple encapsulation mechanisms, such as function closures, object-oriented interfaces, and support for abstract datatypes, in multiple programming languages. [Application]
4. Define and use iterators and other operations on aggregates using idioms most natural in multiple programming languages, including taking functions as arguments. [Application]
5. Write basic algorithms that avoid assigning to mutable state or considering object identity. [Application]
6. Write event handlers for use in reactive systems, such as GUIs. [Application]
7. Explain the relationship between object-oriented inheritance (code-sharing and overriding) and subtyping (the idea of a subtype being usable in a context that expects the supertype). [Knowledge]
8. Explain benefits and limitations of static typing. [Knowledge]
9. For multiple programming languages, identify program properties checked statically and program properties checked dynamically. Use this knowledge when writing and debugging programs. [Application]

10. Distinguish a language definition (what constructs mean) from a particular language implementation (compiler vs. interpreter, run-time representation of data objects, etc.). [Knowledge]
11. Explain how programming language implementations typically organize memory into global data, text, heap, and stack sections and how features such as recursion and memory management map to this memory model. [Knowledge]
12. Reason about memory leaks, dangling-pointer dereferences, and the benefits and limitations of garbage collection. [Application]
13. Process some representation of code for some purpose, such as an interpreter, an expression optimizer, a documentation generator, etc. [Application]

PL/Syntax Analysis

[Elective]

Topics:

- Scanning (lexical analysis) using regular expressions
- Parsing strategies including top-down (e.g., recursive descent, Earley parsing, or LL) and bottom-up (e.g., backtracking or LR) techniques; role of context-free grammars
- Generating scanners and parsers from declarative specifications

Learning outcomes:

1. Use formal grammars to specify the syntax of languages. [Application]
2. Use declarative tools to generate parsers and scanners. [Application]
3. Identify key issues in syntax definitions: ambiguity, associativity, precedence. [Knowledge]

PL/Compiler Semantic Analysis

[Elective]

Topics:

- High-level program representations such as abstract syntax trees
- Scope and binding resolution
- Type checking
- Declarative specifications such as attribute grammars

Learning outcomes:

1. Implement context-sensitive, source-level static analyses such as type-checkers or resolving identifiers to identify their binding occurrences. [Application]

PL/Code Generation

[Elective]

Topics:

- Instruction selection
- Procedure calls and method dispatching
- Register allocation
- Separate compilation; linking
- Instruction scheduling
- Peephole optimization

Learning outcomes:

1. Identify all essential steps for automatically converting source code into assembly or other low-level languages. [Knowledge]
2. Generate the low-level code for calling functions/methods in modern languages. [Application]
3. Discuss opportunities for optimization introduced by naive translation and approaches for achieving optimization. [Knowledge]

PL/Runtime Systems

[Elective]

Topics:

- Target-platform characteristics such as registers, instructions, bytecodes
- Dynamic memory management approaches and techniques: malloc/free, garbage collection (mark-sweep, copying, reference counting), regions (also known as arenas or zones)
- Data layout for objects and activation records
- Just-in-time compilation and dynamic recompilation
- Other features such as class loading, threads, security, etc.

Learning outcomes:

1. Compare the benefits of different memory-management schemes, using concepts such as fragmentation, locality, and memory overhead. [Knowledge]
2. Discuss benefits and limitations of automatic memory management. [Knowledge]
3. Identify the services provided by modern language run-time systems. [Knowledge]
4. Discuss advantages, disadvantages, and difficulties of dynamic recompilation. [Knowledge]

PL/Static Analysis

[Elective]

Topics:

- Relevant program representations, such as basic blocks, control-flow graphs, def-use chains, static single assignment, etc.
- Flow-insensitive analyses, such as type-checking and scalable pointer and alias analyses
- Flow-sensitive analyses, such as forward and backward dataflow analyses
- Path-sensitive analyses, such as software model checking
- Tools and frameworks for defining analyses
- Role of static analysis in program optimization
- Role of static analysis in (partial) verification and bug-finding

Learning outcomes:

1. Define useful static analyses in terms of a conceptual framework such as dataflow analysis. [Application]
2. Communicate why an analysis is correct (sound and terminating). [Application]
3. Distinguish “may” and “must” analyses. [Knowledge]
4. Explain why potential aliasing limits sound program analysis and how alias analysis can help. [Knowledge]
5. Use the results of a static analysis for program optimization and/or partial program correctness. [Application]

PL/Advanced Programming Constructs

[Elective]

Topics:

- Lazy evaluation and infinite streams
- Control Abstractions: Exception Handling, Continuations, Monads
- Object-oriented abstractions: Multiple inheritance, Mixins, Traits, Multimethods
- Metaprogramming: Macros, Generative programming, Model-based development
- Module systems
- String manipulation via pattern-matching
- Dynamic code evaluation (“eval”)
- Language support for checking assertions, invariants, and pre/post-conditions

Learning outcomes:

1. Use various advanced programming constructs and idioms correctly. [Application]
2. Discuss how various advanced programming constructs aim to improve program structure, software quality, and programmer productivity. [Knowledge]
3. Discuss how various advanced programming constructs interact with the definition and implementation of other language features. [Knowledge]

PL/Concurrency and Parallelism

[Elective]

Support for concurrency is a fundamental programming-languages issue with rich material in programming language design, language implementation, and language theory. Due to coverage in other Knowledge Areas, this elective Knowledge Unit aims only to complement the material included elsewhere in the body of knowledge. Courses on programming languages are an excellent place to include a general treatment of concurrency including this other material.

(Cross-reference: PD-Parallel and Distributed Computing)

Topics:

- Constructs for thread-shared variables and shared-memory synchronization
- Actor models
- Futures
- Language support for data parallelism
- Models for passing messages between sequential processes
- Effect of memory-consistency models on language semantics and correct code generation

Learning outcomes:

1. Write correct concurrent programs using multiple programming models. [Application]
2. Explain why programming languages do not guarantee sequential consistency in the presence of data races and what programmers must do as a result. [Knowledge]

PL/Type Systems

[Elective]

Topics:

- Compositional type constructors, such as product types (for aggregates), sum types (for unions), function types, quantified types, and recursive types
- Type checking
- Type safety as preservation plus progress
- Type inference
- Static overloading

Learning outcomes:

1. Define a type system precisely and compositionally. [Application]
2. For various foundational type constructors, identify the values they describe and the invariants they enforce. [Knowledge]
3. Precisely specify the invariants preserved by a sound type system. [Knowledge]

PL/Formal Semantics

[Elective]

Topics:

- Syntax vs. semantics
- Lambda Calculus
- Approaches to semantics: Operational, Denotational, Axiomatic
- Proofs by induction over language semantics
- Formal definitions and proofs for type systems
- Parametricity

Learning outcomes:

1. Give a formal semantics for a small language. [Application]
2. Use induction to prove properties of all (or a well-defined subset of) programs in a language. [Application]
3. Use language-based techniques to build a formal model of a software system. [Application]

PL/Language Pragmatics

[Elective]

Topics:

- Principles of language design such as orthogonality
- Evaluation order, precedence, and associativity
- Eager vs. delayed evaluation
- Defining control and iteration constructs
- External calls and system libraries

Learning outcomes:

1. Discuss the role of concepts such as orthogonality and well-chosen defaults in language design. [Knowledge]
2. Use crisp and objective criteria for evaluating language-design decisions. [Application]

PL/Logic Programming

[Elective]

Topics:

- Clausal representation of data structures and algorithms
- Unification
- Backtracking and search

Learning outcomes:

1. Use a logic language to implement conventional algorithms. [Application]
2. Use a logic language to implement algorithms employing implicit search using clauses and relations. [Application]

Software Development Fundamentals (SDF)

Fluency in the process of software development is a prerequisite to the study of most of computer science. In order to effectively use computers to solve problems, students must be competent at reading and writing programs in multiple programming languages. Beyond programming skills, however, they must be able to design and analyze algorithms, select appropriate paradigms, and utilize modern development and testing tools. This knowledge area brings together those fundamental concepts and skills related to the software development process. As such, it provides a foundation for other software-oriented knowledge areas, most notably Programming Languages, Algorithms and Complexity, and Software Engineering.

It is important to note that this knowledge area is distinct from the old Programming Fundamentals knowledge area from CC2001. Whereas that knowledge area focused exclusively on the programming skills required in an introductory computer science course, this new knowledge area is intended to fill a much broader purpose. It focuses on the entire software development process, identifying those concepts and skills that should be mastered in the first year of a computer science program. This includes the design and simple analysis of algorithms, fundamental programming concepts and data structures, and basic software development methods and tools. As a result of its broader purpose, the Software Development Fundamentals knowledge area includes fundamental concepts and skills that could naturally be listed in other software-oriented knowledge areas (e.g., programming constructs from Programming Languages, simple algorithm analysis from Algorithms & Complexity, simple development methodologies from Software Engineering). Likewise, each of these knowledge areas will contain more advanced material that builds upon the fundamental concepts and skills listed here.

While broader in scope than the old Programming Fundamentals, this knowledge area still allows for considerable flexibility in the design of first-year curricula. For example, the Fundamental Programming Concepts unit identifies only those concepts that are common to all programming paradigms. It is expected that an instructor would select one or more programming paradigms (e.g., object-oriented programming, functional programming, scripting) to illustrate these programming concepts, and would pull paradigm-specific content from the Programming Languages knowledge area to fill out a course. Likewise, an instructor could choose to emphasize formal analysis (e.g., Big-Oh, computability) or design methodologies (e.g., team projects, software life cycle) early, thus integrating hours from the Programming Languages, Algorithms and Complexity, and/or Software Engineering knowledge areas. Thus, the 42-hours of material in this knowledge area should be augmented with core material from one or more of these knowledge areas to form a complete and coherent first-year experience.

When considering the hours allocated to each knowledge unit, it should be noted that these hours reflect the minimal amount of classroom coverage needed to introduce the material. Many software development topics will reappear and be reinforced by later topics (e.g., applying iteration constructs when processing lists). In addition, the mastery of concepts and skills from this knowledge area requires a significant amount of software development experience outside of class.

SDF. Software Development Fundamentals (42 Core-Tier1 hours)

	Core-Tier1 hours	Core-Tier2 hours	Includes Electives
SDF/Algorithms and Design	11		N
SDF/Fundamental Programming Concepts	10		N
SDF/Fundamental Data Structures	12		N
SDF/Development Methods	9		N

SDF/Algorithms and Design

[11 Core-Tier1 hours]

This unit builds the foundation for core concepts in the Algorithms & Complexity knowledge area, most notably in the Basic Analysis and Algorithmic Strategies units.

Topics:

- The concept and properties of algorithms
  - Informal comparison of algorithm efficiency (e.g., operation counts)
- The role of algorithms in the problem-solving process
- Problem-solving strategies
  - Iterative and recursive mathematical functions
  - Iterative and recursive traversal of data structure
  - Divide-and-conquer strategies
- Implementation of algorithms
- Fundamental design concepts and principles
  - Abstraction
  - Program decomposition
  - Encapsulation and information hiding
  - Separation of behavior and implementation

Learning Outcomes:

1. Discuss the importance of algorithms in the problem-solving process. [Knowledge]
2. Discuss how a problem may be solved by multiple algorithms, each with different properties. [Knowledge]
3. Create algorithms for solving simple problems. [Application]
4. Use pseudocode or a programming language to implement, test, and debug algorithms for solving simple problems. [Application]
5. Implement, test, and debug simple recursive functions and procedures. [Application]
6. Determine when a recursive solution is appropriate for a problem. [Evaluation]
7. Implement a divide-and-conquer algorithm for solving a problem. [Application]
8. Apply the techniques of decomposition to break a program into smaller pieces. [Application]
9. Identify the data components and behaviors of multiple abstract data types. [Application]
10. Implement a coherent abstract data type, with loose coupling between components and behaviors. [Application]
11. Identify the relative strengths and weaknesses among multiple designs or implementations for a problem. [Evaluation]

SDF/Fundamental Programming Concepts

[10 Core-Tier1 hours]

This unit builds the foundation for core concepts in the Programming Languages knowledge area, most notably in the paradigm-specific units: Object-Oriented Programming, Functional Programming, and Event-Driven & Reactive Programming.

Topics:

- Basic syntax and semantics of a higher-level language
- Variables and primitive data types (e.g., numbers, characters, Booleans)
- Expressions and assignments
- Simple I/O
- Conditional and iterative control structures

- Functions and parameter passing
- The concept of recursion

Learning Outcomes:

1. Analyze and explain the behavior of simple programs involving the fundamental programming constructs covered by this unit. [Evaluation]
2. Identify and describe uses of primitive data types. [Knowledge]
3. Write programs that use each of the primitive data types. [Application]
4. Modify and expand short programs that use standard conditional and iterative control structures and functions. [Application]
5. 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, the definition of functions, and parameter passing. [Application]
6. Choose appropriate conditional and iteration constructs for a given programming task. [Evaluation]
7. Describe the concept of recursion and give examples of its use. [Knowledge]
8. Identify the base case and the general case of a recursively-defined problem. [Evaluation]

SDF/Fundamental Data Structures

[12 Core-Tier1 hours]

This unit builds the foundation for core concepts in the Algorithms & Complexity knowledge area, most notably in the Fundamental Data Structures & Algorithms and Basic Computability & Complexity units.

Topics:

- Arrays
- Records/structs (heterogeneous aggregates)
- Strings and string processing
- Stacks, queues, priority queues, sets & maps
- References and aliasing
- Simple linked structures
- Strategies for choosing the appropriate data structure

Learning Outcomes:

1. Discuss the appropriate use of built-in data structures. [Knowledge]
2. Describe common applications for each data structure in the topic list. [Knowledge]
3. Compare alternative implementations of data structures with respect to performance. [Evaluation]
4. Write programs that use each of the following data structures: arrays, strings, linked lists, stacks, queues, sets, and maps. [Application]
5. Compare and contrast the costs and benefits of dynamic and static data structure implementations. [Evaluation]
6. Choose the appropriate data structure for modeling a given problem. [Evaluation]

SDF/Development Methods

[9 Core-Tier1 hours]

This unit builds the foundation for core concepts in the Software Engineering knowledge area, most notably in the Software Design and Software Processes units.

Topics:

- Program correctness
- The concept of a specification
- Defensive programming (e.g. secure coding, exception handling)
- Code reviews
- Testing fundamentals and test-case generation
- Test-driven development
- The role and the use of contracts, including pre- and post-conditions
- Unit testing
- Modern programming environments
- Programming using library components and their APIs
- Debugging strategies
- Documentation and program style

Learning Outcomes:

1. Explain why the creation of correct program components is important in the production of quality software. [Knowledge]
2. Identify common coding errors that lead to insecure programs (e.g., buffer overflows, memory leaks, malicious code) and apply strategies for avoiding such errors. [Application]
3. Conduct a personal code review (focused on common coding errors) on a program component using a provided checklist. [Application]
4. Contribute to a small-team code review focused on component correctness. [Application]
5. Describe how a contract can be used to specify the behavior of a program component. [Knowledge]
6. Create a unit test plan for a medium-size code segment. [Application]
7. Apply a variety of strategies to the testing and debugging of simple programs. [Application]
8. Construct, execute and debug programs using a modern IDE (e.g., Visual Studio or Eclipse) and associated tools such as unit testing tools and visual debuggers. [Application]
9. Construct and debug programs using the standard libraries available with a chosen programming language. [Application]
10. Apply consistent documentation and program style standards that contribute to the readability and maintainability of software. [Application]

Software Engineering (SE)

In every computing application domain, professionalism, quality, schedule, and cost are critical to producing software systems. Because of this, the elements of software engineering are applicable to developing software in all areas of computing. A wide variety of software engineering practices have been developed and utilized since the need for a discipline of software engineering was first recognized. Many trade-offs between these different practices have also been identified. Practicing software engineers have to select and apply appropriate techniques and practices to a given development effort to maximize value. To learn how to do this, they study the elements of software engineering.

Software engineering is the discipline concerned with the application of theory, knowledge, and practice to effectively and efficiently build reliable software systems that satisfy the requirements of customers and users. This discipline is applicable to small, medium, and large-scale systems. It encompasses all phases of the lifecycle of a software system, including requirements elicitation, analysis and specification; design; construction; verification and validation; deployment; and operation and maintenance. Whether small or large, following a traditional disciplined development process, an agile approach, or some other method, software engineering is concerned with the best way to build good software systems.

Software engineering uses engineering methods, processes, techniques, and measurements. It benefits from the use of tools for managing software development; analyzing and modeling software artifacts; assessing and controlling quality; and for ensuring a disciplined, controlled approach to software evolution and reuse. The software engineering toolbox has evolved over the years. For instance, the use of contracts, with requires and ensure clauses and class invariants, is one good practice that has become more common. Software development, which can involve an individual developer or a team or teams of developers, requires choosing the most appropriate tools, methods, and approaches for a given development environment.

Students and instructors need to understand the impacts of specialization on software engineering approaches. For example, specialized systems include:

- Real time systems
- Client-server systems
- Distributed systems
- Parallel systems
- Web-based systems
- High integrity systems
- Games
- Mobile computing
- Domain specific software (e.g., scientific computing or business applications)

Issues raised by each of these specialized systems demand specific treatments in each phase of software engineering. Students must become aware of the differences between general software engineering techniques and principles and the techniques and principles needed to address issues specific to specialized systems.

An important effect of specialization is that different choices of material may need to be made when teaching applications of software engineering, such as between different process models, different approaches to modeling systems, or different choices of techniques for carrying out any of the key activities. This is reflected in the assignment of core and elective material, with the core topics and learning outcomes focusing on the principles underlying the various choices, and the details of the various alternatives from which the choices have to be made being assigned to the elective material.

Another division of the practices of software engineering is between those concerned with the fundamental need to develop systems that implement correctly the functionality that is required for them, and those concerned with other qualities for systems and the trade-offs needed to balance these qualities. This division too is reflected in the assignment of core and elective material, so that topics and learning outcomes concerned with the basic methods for developing such system are assigned to the core, and those that are concerned with other qualities and trade-offs between them are assigned to the elective material.

In general, students learn best at the application level much of the material defined in the SE KA by participating in a project. Such projects should require students to work on a team to develop a software system through as much of its lifecycle as is possible. Much of software engineering is devoted to effective communication among team members and stakeholders. Utilizing project teams, projects can be sufficiently challenging to require the use of effective software engineering techniques and that students develop and practice their communication skills. While organizing and running effective projects within the academic framework can be challenging, the best way to learn to apply software engineering theory and knowledge is in the practical environment of a project. The minimum hours specified for some knowledge units in this document may appear insufficient to accomplish associated application-level learning outcomes. It should be understood that these outcomes are to be achieved through project experience that may even occur later in the curriculum than when the topics within the knowledge unit are introduced.

Note: The SDF/Development Methods knowledge unit includes 9 Core-Tier1 hours that constitute an introduction to certain aspects of software engineering. The knowledge units, topics and core hour specifications in this document must be understood as assuming previous exposure to the material described in SDF/Development Methods.

SE. Software Engineering (6 Core-Tier1 hours; 21 Core-Tier2 hours)

	Core-Tier1 hours	Core-Tier2 hours	Includes Electives
SE/Software Processes	1	2	Y
SE/Software Project Management		3	Y
SE/Tools and Environments		2	N
SE/Requirements Engineering	1	3	Y
SE/Software Design	4	4	Y
SE/Software Construction		2	Y
SE/Software Verification Validation		3	Y
SE/Software Evolution		1	Y
SE/Formal Methods			Y
SE/Software Reliability		1	Y

SE/Software Processes

[1 Core-Tier1 hours; 2 Core-Tier2 hours]

Topics:

[Core-Tier1]

- Systems level considerations, i.e., the interaction of software with its intended environment
- Phases of software life-cycles
- Programming in the large vs. individual programming

[Core-Tier2]

- Software process models (e.g., waterfall, incremental, agile)

[Elective]

- Software quality concepts
- Process improvement
- Software process capability maturity models
- Software process measurements

Learning Outcomes:

[Core-Tier1]

1. Describe how software can interact with and participate in various systems including information management, embedded, process control, and communications systems. [Knowledge]
2. Differentiate among the phases of software development. [Knowledge]
3. Explain the concept of a software life cycle and provide an example, illustrating its phases including the deliverables that are produced. [Knowledge]
4. Describe how programming in the large differs from individual efforts with respect to understanding a large code base, code reading, understanding builds, and understanding context of changes. [Knowledge]

[Core-Tier2]

1. Describe the difference between principles of the waterfall model and models using iterations. [Knowledge]
2. Compare several common process models with respect to their value for development of particular classes of software systems taking into account issues such as requirement stability, size, and non-functional characteristics. [Application]

[Elective]

1. Define software quality and describe the role of quality assurance activities in the software process. [Knowledge]
2. Describe the intent and fundamental similarities among process improvement approaches. [Knowledge]
3. Compare several process improvement models such as CMM, CMMI, CQI, Plan-Do-Check-Act, or ISO9000. [Knowledge]
4. Use a process improvement model such as PSP to assess a development effort and recommend approaches to improvement. [Application]
5. Explain the role of process maturity models in process improvement. [Knowledge]
6. Describe several process metrics for assessing and controlling a project. [Knowledge]
7. Use project metrics to describe the current state of a project. [Application]

SE/Software Project Management

[3 Core-Tier2 hours]

Topics:

[Core-Tier2]

- Risk
  - The role of risk in the life cycle
  - Risk categories including security, safety, market, financial, technology, people, quality, structure and process
  - Risk identification
  - Risk tolerance (e.g., risk-adverse, risk-neutral, risk-seeking)
  - Risk planning
  - Risk removal, reduction and control
- Team participation
  - Team processes including responsibilities for tasks, meeting structure, and work schedule
  - Roles and responsibilities in a software team
  - Team conflict resolution
  - Risks associated with virtual teams (communication, perception, structure)
- Effort Estimation (at the personal level)

[Elective]

- Team management
  - Team organization and decision-making
  - Role identification and assignment
  - Individual and team performance assessment
- Project management
  - Scheduling and tracking
  - Project management tools
  - Cost/benefit analysis
- Software measurement and estimation techniques
- Software quality assurance and the role of measurements
- Principles of risk management
- Risk analysis and evaluation
- System-wide approach to risk including hazards associated with tools

Learning Outcomes:

[Core-Tier2]

1. List several examples of software risks. [Knowledge]
2. Describe the impact of risk in a software development life cycle. [Knowledge]
3. Describe different categories of risk in software systems. [Knowledge]
4. Describe the impact of risk tolerance on the software development process. [Application]
5. Identify risks and describe approaches to managing risk (avoidance, acceptance, transference, mitigation), and characterize the strengths and shortcomings of each. [Knowledge]
6. Explain how risk affects decisions in the software development process. [Application]
7. Identify behaviors that contribute to the effective functioning of a team. [Knowledge]
8. Create and follow an agenda for a team meeting. [Application]
9. Identify and justify necessary roles in a software development team. [Application]
10. Understand the sources, hazards, and potential benefits of team conflict. [Application]
11. Apply a conflict resolution strategy in a team setting. [Application]
12. Use an *ad hoc* method to estimate software development effort (e.g., time) and compare to actual effort required. [Application]

[Elective]

1. Identify security risks for a software system. [Application]
2. Demonstrate through involvement in a team project the central elements of team building and team management. [Application]
3. Identify several possible team organizational structures and team decision-making processes. [Knowledge]
4. Create a team by identifying appropriate roles and assigning roles to team members. [Application]
5. Assess and provide feedback to teams and individuals on their performance in a team setting. [Application]
6. Prepare a project plan for a software project that includes estimates of size and effort, a schedule, resource allocation, configuration control, change management, and project risk identification and management. [Application]
7. Track the progress of a project using appropriate project metrics. [Application]
8. Compare simple software size and cost estimation techniques. [Application]
9. Use a project management tool to assist in the assignment and tracking of tasks in a software development project. [Application]
10. Demonstrate a systematic approach to the task of identifying hazards and risks in a particular situation. [Application]
11. Apply the basic principles of risk management in a variety of simple scenarios including a security situation. [Application]
12. Conduct a cost/benefit analysis for a risk mitigation approach. [Application]
13. Identify and analyze some of the risks for an entire system that arise from aspects other than the software.

SE/Tools and Environments

[2 Core-Tier2 hours]

Topics:

[Core-Tier2]

- Software configuration management and version control; release management
- Requirements analysis and design modeling tools
- Testing tools including static and dynamic analysis tools
- Programming environments that automate parts of program construction processes (e.g., automated builds)
- Tool integration concepts and mechanisms

Learning Outcomes:

[Core-Tier2]

1. Describe the difference between centralized and distributed software configuration management. [Knowledge]
2. Identify configuration items and use a source code control tool in a small team-based project. [Application]
3. Describe the issues that are important in selecting a set of tools for the development of a particular software system, including tools for requirements tracking, design modeling, implementation, build automation, and testing. [Knowledge]
4. Demonstrate the capability to use software tools in support of the development of a software product of medium size. [Application]

SE/Requirements Engineering

[1 Core-Tier1 hour; 3 Core-Tier2 hours]

Topics:

[Core-Tier1]

- Fundamentals of software requirements elicitation and modeling

[Core-Tier2]

- Properties of requirements including consistency, validity, completeness, and feasibility
- Software requirements elicitation
- Describing functional requirements using, for example, use cases or users stories
- Non-functional requirements and their relationship to software quality
- Describing system data using, for example, class diagrams or entity-relationship diagrams
- Evaluation and use of requirements specifications

[Elective]

- Requirements analysis modeling techniques
- Acceptability of certainty / uncertainty considerations regarding software / system behavior
- Prototyping
- Basic concepts of formal requirements specification
- Requirements specification
- Requirements validation
- Requirements tracing

Learning Outcomes:

[Core-Tier1]

1. Describe the fundamental challenges of and common techniques used for requirements elicitation. [Knowledge]
2. Interpret a given requirements model for a simple software system. [Knowledge]

[Core-Tier2]

1. Conduct a review of a set of software requirements to determine the quality of the requirements with respect to the characteristics of good requirements. [Application]
2. List the key components of a use case or similar description of some behavior that is required for a system and discuss their role in the requirements engineering process. [Knowledge]
3. List the key components of a class diagram or similar description of the data that a system is required to handle. [Knowledge]
4. Identify both functional and non-functional requirements in a given requirements specification for a software system. [Application]

[Elective]

1. Apply key elements and common methods for elicitation and analysis to produce a set of software requirements for a medium-sized software system. [Application]
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 [Application]
3. Translate into natural language a software requirements specification (e.g., a software component contract) written in a formal specification language. [Application]
4. Create a prototype of a software system to mitigate risk in requirements. [Application]

5. Differentiate between forward and backward tracing and explain their roles in the requirements validation process. [Knowledge]

SE/Software Design

[4 Core-Tier1 hours; 4 Core-Tier2 hours]

Topics:

[Core-Tier1]

- Overview of design paradigms
- System design principles: divide and conquer (architectural design and detailed design), separation of concerns, information hiding, coupling and cohesion, re-use of standard structures.
- Appropriate models of software designs, including structure and behavior.
- Software architecture concepts

[Core-Tier2]

- Design Paradigms such as structured design (top-down functional decomposition), object-oriented analysis and design, event driven design, component-level design, data-structured centered, aspect oriented, function oriented, service oriented.
- Relationships between requirements and designs: transformation of models, design of contracts.
- Architectural design: standard architectures (e.g. client-server, n-layer, transform centered, pipes-and-filters, etc).
- Refactoring designs and the use of design patterns.
- The use of components in design: component selection, design, adaptation and assembly of components, components and patterns, components and objects, (for example, build a GUI using a standard widget set).

[Elective]

- Internal design qualities, and models for them: efficiency and performance, redundancy and fault tolerance, traceability of requirements.
- External design qualities, and models for them: functionality, reliability, performance and efficiency, usability, maintainability, portability.
- Measurement and analysis of design quality.
- Tradeoffs between different aspects of quality.
- Application frameworks.
- Middleware: the object-oriented paradigm within middleware, object request brokers and marshalling, transaction processing monitors, workflow systems.

Learning Outcomes:

[Core-Tier1]

1. Articulate design principles including separation of concerns, information hiding, coupling and cohesion, and encapsulation. [Knowledge]
2. Use a design paradigm to design a simple software system, and explain how system design principles have been applied in this design. [Application]
3. Construct models of the design of a simple software system that are appropriate for the paradigm used to design it. [Application]
4. For the design of a simple software system within the context of a single design paradigm, describe the software architecture of that system. [Knowledge]
5. Within the context of a single design paradigm, describe one or more design patterns that could be applicable to the design of a simple software system. [Knowledge]

6. Given a high-level design, identify the software architecture by differentiating among common software architectures such as 3-tier, pipe-and-filter, and client-server. [Knowledge]

[Core-Tier2]

1. For a simple system suitable for a given scenario, discuss and select an appropriate design paradigm. [Application]
2. Create appropriate models for the structure and behavior of software products from their requirements specifications. [Application]
3. Explain the relationships between the requirements for a software product and the designed structure and behavior, in terms of the appropriate models and transformations of them. [Evaluation]
4. Apply simple examples of patterns in a software design. [Application]
5. Investigate the impact of software architectures selection on the design of a simple system.
6. Select suitable components for use in the design of a software product. [Application]
7. Explain how suitable components might need to be adapted for use in the design of a software product. [Knowledge].
8. Design a contract for a typical small software component for use in a given system. [Application]

[Elective]

1. Discuss and select appropriate software architecture for a simple system suitable for a given scenario. [Application]
2. Apply models for internal and external qualities in designing software components to achieve an acceptable tradeoff between conflicting quality aspects. [Application]
3. Analyze a software design from the perspective of a significant internal quality attribute. [Evaluation]
4. Analyze a software design from the perspective of a significant external quality attribute. [Evaluation]
5. Explain the role of objects in middleware systems and the relationship with components. [Knowledge]
6. Apply component-oriented approaches to the design of a range of software, such as using components for concurrency and transactions, for reliable communication services, for database interaction including services for remote query and database management, or for secure communication and access. [Application]

SE/Software Construction

[2 Core-Tier2 hours]

Topics:

[Core-Tier2]

- Coding practices: techniques, idioms/patterns, mechanisms for building quality programs
  - Defensive coding practices
  - Secure coding practices
  - Using exception handling mechanisms to make programs more robust, fault-tolerant
- Coding standards
- Integration strategies

[Elective]

- Robust And Security Enhanced Programming
  - Defensive programming
  - Principles of secure design and coding:
  - Principle of least privilege
  - Principle of fail-safe defaults
  - Principle of psychological acceptability
- Potential security problems in programs
  - Buffer and other types of overflows
  - Race conditions
  - Improper initialization, including choice of privileges
  - Checking input
  - Assuming success and correctness
  - Validating assumptions
- Documenting security considerations in using a program

Learning Outcomes:

[Core-Tier2]

1. Describe techniques, coding idioms and mechanisms for implementing designs to achieve desired properties such as reliability, efficiency, and robustness. [Knowledge]
2. Build robust code using exception handling mechanisms. [Application]
3. Describe secure coding and defensive coding practices. [Knowledge]
4. Select and use a defined coding standard in a small software project. [Application]
5. Compare and contrast integration strategies including top-down, bottom-up, and sandwich integration. [Knowledge]

[Elective]

1. Rewrite a simple program to remove common vulnerabilities, such as buffer overflows, integer overflows and race conditions
2. State and apply the principles of least privilege and fail-safe defaults.
3. Write a simple library that performs some non-trivial task and will not terminate the calling program regardless of how it is called

SE/Software Verification Validation

[3 Core-Tier2 hours]

Topics:

[Core-Tier2]

- Verification and validation concepts
- Inspections, reviews, audits
- Testing types, including human computer interface, usability, reliability, security, conformance to specification
- Testing fundamentals
- Unit, integration, validation, and system testing
- Test plan creation and test case generation
- Black-box and white-box testing techniques
- Defect tracking
- Testing parallel and distributed systems

[Elective]

- Static approaches and dynamic approaches to verification
- Regression testing
- Test-driven development
- Validation planning; documentation for validation
- Object-oriented testing; systems testing
- Verification and validation of non-code artifacts (documentation, help files, training materials)
- Fault logging, fault tracking and technical support for such activities
- Fault estimation and testing termination including defect seeding

Learning Outcomes:

[Core-Tier2]

1. Distinguish between program validation and verification. [Knowledge]
2. Describe the role that tools can play in the validation of software. [Knowledge]
3. Undertake, as part of a team activity, an inspection of a medium-size code segment. [Application]
4. Describe and distinguish among the different types and levels of testing (unit, integration, systems, and acceptance). [Knowledge]
5. Describe techniques for identifying significant test cases for unit, integration, and system testing. [Knowledge]
6. Use a defect tracking tool to manage software defects in a small software project. [Application]
7. Describe the issues and approaches to testing distributed and parallel systems. [Knowledge]

[Elective]

1. Create, evaluate, and implement a test plan for a medium-size code segment. [Application]
2. Compare static and dynamic approaches to verification. [Knowledge]
3. Discuss the issues involving the testing of object-oriented software. [Application]
4. Describe techniques for the verification and validation of non-code artifacts. [Knowledge]
5. Describe approaches for fault estimation. [Knowledge]
6. Estimate the number of faults in a small software application based on fault density and fault seeding. [Application]
7. Conduct an inspection or review of software source code for a small or medium sized software project. [Application]

SE/Software Evolution

[1 Core-Tier2 hour]

Topics:

[Core-Tier2]

- Software development in the context of large, pre-existing code bases
- Software evolution
- Characteristics of maintainable software
- Reengineering systems
- Software reuse

Learning Outcomes:

[Core-Tier2]

1. Identify the principal issues associated with software evolution and explain their impact on the software life cycle. [Knowledge]
2. Discuss the challenges of evolving systems in a changing environment. [Knowledge]
3. Outline the process of regression testing and its role in release management. [Application]
4. Discuss the advantages and disadvantages of software reuse. [Knowledge]

[Elective]

1. Estimate the impact of a change request to an existing product of medium size. [Application]
2. Identify weaknesses in a given simple design, and removed them through refactoring. [Application]

SE/Formal Methods

[Elective]

The topics listed below have a strong dependency on core material from the Discrete Structures area, particularly knowledge units DS/Functions Relations And Sets, DS/Basic Logic and DS/Proof Techniques.

***Topics*:**

- Role of formal specification and analysis techniques in the software development cycle
- Program assertion languages and analysis approaches (including languages for writing and analyzing pre- and post-conditions, such as OCL, JML)
- Formal approaches to software modeling and analysis
- Model checkers
- Model finders
- Tools in support of formal methods

Learning Outcomes:

1. Describe the role formal specification and analysis techniques can play in the development of complex software and compare their use as validation and verification techniques with testing. [Knowledge]

2. Apply formal specification and analysis techniques to software designs and programs with low complexity. [Application]
3. Explain the potential benefits and drawbacks of using formal specification languages. [Knowledge]
4. Create and evaluate program assertions for a variety of behaviors ranging from simple through complex. [Application]
5. Using a common formal specification language, formulate the specification of a simple software system and derive examples of test cases from the specification. [Application]

SE/Software Reliability

[1 Core-Tier2]

Topics:

[Core-Tier2]

- Software reliability engineering concepts
- Software reliability, system reliability and failure behavior (cross-reference SF9/Reliability Through Redundancy)
- Fault lifecycle concepts and techniques

[Elective]

- Software reliability models
- Software fault tolerance techniques and models
- Software reliability engineering practices
- Measurement-based analysis of software reliability

Learning Outcomes:

[Core-Tier2]

1. Explain the problems that exist in achieving very high levels of reliability. [Knowledge]
2. Describe how software reliability contributes to system reliability [Knowledge]
3. List approaches to minimizing faults that can be applied at each stage of the software lifecycle. [Knowledge]

[Elective]

1. Compare the characteristics of three different reliability modeling approaches. [Knowledge]
2. Demonstrate the ability to apply multiple methods to develop reliability estimates for a software system. [Application]
3. Identify methods that will lead to the realization of a software architecture that achieves a specified reliability level of reliability. [Application]
4. Identify ways to apply redundancy to achieve fault tolerance for a medium-sized application. [Application]

Systems Fundamentals (SF)

The underlying hardware and software infrastructure upon which applications are constructed is collectively described by the term "computer systems." Computer systems broadly span the sub-disciplines of operating systems, parallel and distributed systems, communications networks, and computer architecture. Traditionally, these areas are taught in a non-integrated way through independent courses. However these sub-disciplines increasingly share important common fundamental concepts within their respective cores. These include computational paradigms, parallelism, cross-layer communications, state and state transition, resource allocation and scheduling, and so on. This knowledge area presents an integrative cross-layer view of these fundamental concepts in a unified albeit simplified fashion, providing a common foundation for the different specialized mechanisms and policies appropriate to the particular knowledge areas that it underlies. An organizing principle is “programming for performance”: what does a programmer need to know about the underlying system in order to achieve high performance in an application being developed.

SF. Systems Fundamentals [18 core Tier 1, 9 core Tier 2 hours, 27 total]

	Core-Tier 1 hours	**Core-Tier 2 hours**	**Includes Electives**
SF/Computational Paradigms	**3**		
SF/Cross-Layer Communications	**3**		
SF/State-State Transition-State Machines	**6**		
SF/System Support for Parallelism	**3**		
SF/Performance	**3**		
SF/Resource Allocation and Scheduling		**2**	
SF/Proximity		**3**	
SF/Virtualization and Isolation		**2**	
SF/Reliability through Redundancy		**2**	

SF/Computational Paradigms

[3 Core-Tier 1 hours]

[Cross-reference PD/parallelism fundamentals: The view presented here is the multiple representations of a system across layers, from hardware building blocks to application components, and the parallelism available in each representation; PD/parallelism fundamentals focuses on the application structuring concepts for parallelism.]

Topics:

- A computing system as a layered collection of representations
- Basic building blocks and components of a computer (gates, flip-flops, registers, interconnections; Datapath + Control + Memory)
- Hardware as a computational paradigm: Fundamental logic building blocks (logic gates, flip-flops, counters, registers, PL); Logic expressions, minimization, sum of product forms
- Application-level sequential processing: single thread [xref PF/]
- Simple application-level parallel processing: request level (web services/client-server/distributed), single thread per server, multiple threads with multiple servers
- Basic concept of pipelining, overlapped processing stages
- Basic concept of scaling: going faster vs. handling larger problems

Learning Outcomes:

1. List commonly encountered patterns of how computations are organized [Knowledge].
2. Describe the basic building blocks of computers and their role in the historical development of computer architecture [Knowledge].
3. Articulate the differences between single thread vs. multiple thread, single server vs. multiple server models, motivated by real world examples (e.g., cooking recipes, lines for multiple teller machines, couple shopping for food, wash-dry-fold, etc.) [Knowledge].
4. Articulate the concept of strong vs. weak scaling, i.e., how performance is affected by scale of problem vs. scale of resources to solve the problem. This can be motivated by the simple, real-world examples [Knowledge].
5. Design and simulate a simple logic circuit using the fundamental building blocks of logic design [Application].
6. Write a simple sequential problem and a simple parallel version of the same program [Application].
7. Evaluate performance of simple sequential and parallel versions of a program with different problem sizes, and be able to describe the speed-ups achieved [Evaluation].

SF/Cross-Layer Communications

[3 Core-Tier 1 hours]

Topics:

- Programming abstractions, interfaces, use of libraries
- Distinction between application and OS services, remote procedure call
- Interactions between applications and virtual machines
- Reliability

Learning Outcomes:

1. Describe how computing systems are constructed of layers upon layers, based on separation of concerns, with well-defined interfaces, hiding details of low layers from the higher layers [knowledge].
2. Recognize that hardware, VM, OS, application are just additional layers of interpretation/processing [knowledge].
3. Describe the mechanisms of how errors are detected, signaled back, and handled through the layers [knowledge]
4. Construct a simple program using methods of layering, error detection and recovery, and reflection of error status across layers [application].
5. Find bugs in a layered program by using tools for program tracing, single stepping, and debugging [evaluation].

SF/State-State Transition-State Machines

[6 Core-Tier 1 hours]

[Cross-reference AL/Basic Computability and Complexity, OS/state and state diagrams, NC/protocols]

Topics:

- Digital vs. analog/discrete vs. continuous systems
- Simple logic gates, logical expressions, Boolean logic simplification
- Clocks, state, sequencing
- Combinational Logic, Sequential Logic, Registers, Memories
- Computers and Network Protocols as examples of State Machines

Learning Outcomes:

1. Describe computations as a system with a known set of configurations, and a byproduct of the computation is to transition from one unique configuration (state) to another (state) [Knowledge].
2. Recognize the distinction between systems whose output is only a function of their input (Combinational) and those with memory or history (Sequential) [Knowledge].
3. Describe a computer as a state machine that interprets machine instructions [Knowledge].
4. Explain how a program or network protocol can also be expressed as a state machine, and that alternative representations for the same computation can exist [Knowledge].
5. Develop state machine descriptions for simple problem statement solutions (e.g., traffic light sequencing, pattern recognizers) [Application].
6. Derive time-series behavior of a state machine from its state machine representation [Evaluation].

SF/System Support for Parallelism

[3 Core-Tier1 hours]

[Cross-reference: PD/Parallelism Fundamentals]

Topics:

- Execution and runtime models that distinguish Sequential vs. Parallel processing
- System organizations that support Request and Task parallelism and other parallel processing paradigms, such as Client-Server/Web Services, Thread parallelism(Fork-Join), and Pipelining
- Multicore architectures and hardware support for parallelism

***Learning Outcomes**:*

1. For a given program, distinguish between its sequential and parallel execution, and the performance implications thereof [knowledge].
2. Demonstrate on an execution time line that parallel events and operations can take place simultaneously (i.e., at the same time). Explain how work can be performed in less elapsed time if this can be exploited [knowledge].
3. Explain other uses of parallelism, such as for reliability/redundancy of execution [knowledge].
4. Define the differences between the concepts of Instruction Parallelism, Data Parallelism, Thread Parallelism/Multitasking, Task/Request Parallelism.
5. Write a simple parallel program in more than one paradigm so as to be able to compare and contrast ease of expression and performance in solving a given problem [application].
6. Use performance tools to measure speed-up achieved by parallel programs in terms of both problem size and number of resources [evaluation].

SF/Performance

[3 Core-Tier 1 hours]

[Cross-reference PD/Parallel Performance]

***Topics**:*

- Figures of performance merit (e.g., speed of execution, energy consumption, bandwidth vs. latency, resource cost)
- Benchmarks (e.g., SPEC) and measurement methods
- CPI equation (Execution time = # of instructions * cycles/instruction * time/cycle) as tool for understanding tradeoffs in the design of instruction sets, processor pipelines, and memory system organizations.
- Amdahl's Law: the part of the computation that cannot be sped up limits the effect of the parts that can

***Learning Outcomes**:*

1. Explain how the components of system architecture contribute to improving its performance [Knowledge].
2. Describe Amdahl's law and its implications for parallel system speed-up when limited by sequential portions, e.g., in processing pipelines [Knowledge].
3. Benchmark a parallel program with different data sets in order to iteratively improve its performance [Application].
4. Use software tools to profile and measure program performance [Evaluation].

SF/Resource Allocation and Scheduling

[2 Core-Tier 2 hours]

***Topics**:*

- Kinds of resources: processor share, memory, disk, net bandwidth
- Kinds of scheduling: first-come, priority
- Advantages of fair scheduling, preemptive scheduling

***Learning Outcomes**:*

1. Define how finite computer resources (e.g., processor share, memory, storage and network bandwidth) are managed by their careful allocation to existing entities [Knowledge].

2. Describe the scheduling algorithms by which resources are allocated to competing entities, and the figures of merit by which these algorithms are evaluated, such as fairness [Knowledge].
3. Implement simple scheduling algorithms [Application].
4. Measure figures of merit of different scheduler implementations [Evaluation].

SF/Proximity

[3 Core-Tier 2 hours]

[Cross-reference: AR/Memory Management, OS/VM/Virtual Memory]

Topics:

- Speed of light and computers (one foot per nanosecond vs. one GHz clocks)
- Latencies in computer systems: memory vs. disk latencies vs. across the network memory
- Caches, spatial and temporal locality, in processors and systems
- Elementary introduction into the processor memory hierarchy: registers and multi-level caches, and the formula for average memory access time

Learning Outcomes:

1. Explain the importance of locality in determining performance [Knowledge].
2. Describe why things that are close in space take less time to access [Knowledge].
3. Calculate average memory access time and describe the tradeoffs in memory hierarchy performance in terms of capacity, miss/hit rate, and access time [Evaluation].

SF/Virtualization and Isolation

[2 Core-Tier 2 hours]

Topics:

- Rationale for protection and predictable performance
- Levels of indirection, illustrated by virtual memory for managing physical memory resources
- Methods for implementing virtual memory and virtual machines

Learning Outcomes:

1. Explain why it is important to isolate and protect the execution of individual programs and environments that share common underlying resources, including the processor, memory, storage, and network access [Knowledge].
2. Describe how the concept of indirection can create the illusion of a dedicated machine and its resources even when physically shared among multiple programs and environments [Knowledge].
3. Measure the performance of two application instances running on separate virtual machines, and determine the effect of performance isolation [Evaluation].

SF/Reliability through Redundancy

[2 Core-Tier 2 hours]

Topics:

- Distinction between bugs and faults, and how they arise in hardware vs. software
- How errors increase the longer the distance between the communicating entities; the end-to-end principle as it applies to systems and networks
- Redundancy through check and retry
- Redundancy through redundant encoding (error correcting codes, CRC/Cyclic Redundancy Codes, FEC/Forward Error Correction)
- Duplication/mirroring/replicas

Learning Outcomes:

1. Explain the distinction between program errors, system errors, and hardware faults (e.g., bad memory) and exceptions (e.g., attempt to divide by zero) [Knowledge].
2. Articulate the distinction between detecting, handling, and recovering from faults, and the methods for their implementation [Knowledge].
3. Describe the role of error correcting codes in providing error checking and correction techniques in memories, storage, and networks [Knowledge].
4. Apply simple algorithms for exploiting redundant information for the purposes of data correction [Application].
5. Compare different error detection and correction methods for their data overhead, implementation complexity, and relative execution time for encoding, detecting, and correcting errors [Evaluation].

Social and Professional Practice (SP)

While technical issues are central to the computing curriculum, they do not constitute a complete educational program in the field. Students must also be exposed to the larger societal context of computing to develop an understanding of the relevant social, ethical and professional issues. This need to incorporate the study of these non-technical issues into the ACM curriculum was formally recognized in 1991, as can be seen from the following excerpt [Tucker91]:

> *Undergraduates also need to understand the basic cultural, social, legal, and ethical issues inherent in the discipline of computing. They should understand where the discipline has been, where it is, and where it is heading. They should also understand their individual roles in this process, as well as appreciate the philosophical questions, technical problems, and aesthetic values that play an important part in the development of the discipline.*
>
> *Students also need to develop the ability to ask serious questions about the social impact of computing and to evaluate proposed answers to those questions. Future practitioners must be able to anticipate the impact of introducing a given product into a given environment. Will that product enhance or degrade the quality of life? What will the impact be upon individuals, groups, and institutions?*
>
> *Finally, students need to be aware of the basic legal rights of software and hardware vendors and users, and they also need to appreciate the ethical values that are the basis for those rights. Future practitioners must understand the responsibility that they will bear, and the possible consequences of failure. They must understand their own limitations as well as the limitations of their tools. All practitioners must make a long-term commitment to remaining current in their chosen specialties and in the discipline of computing as a whole.*

As technological advances continue to significantly impact the way we live and work, the critical importance of these social and professional issues continues to increase; new computer-based products and venues pose ever more challenging problems each year. It is our students who must enter the workforce and academia with intentional regard for the identification and resolution of these problems.

Computer science educators may opt to deliver this core and elective material in stand-alone courses, integrated into traditional technical and theoretical courses, or as special units in capstone and professional practice courses. The material in this knowledge area is best covered through a combination of one required course along with short modules in other courses. On the one hand, some units listed as core-tier 1—in particular, Social Context, Analytical Tools, Professional Ethics, and Intellectual Property—do not readily lend themselves to being covered in other traditional courses. Without a standalone course, it is difficult to cover these topics appropriately. On the other hand, if ethical considerations are covered only in the standalone course and not "in context," it will reinforce the false notion that technical processes are void of ethical issues. Thus it is important that several traditional courses include modules that analyze ethical considerations in the context of the technical subject matter of the course. Courses in areas such as software engineering, databases, computer networks, and introduction to computing provide obvious context for analysis of ethical issues. However, an ethics-related module could be developed for almost any course in the curriculum. It would be explicitly against the spirit of the recommendations to have only a standalone course. Running through all of the issues in this area is the need to speak to the computer practitioner's responsibility to proactively address these issues by both moral and technical actions. The ethical issues discussed in any class should be directly related to and arise naturally from the subject matter of that class. Examples include a discussion in the database course of data aggregation or data mining, or a discussion in the software engineering course of the potential conflicts between obligations to the customer and obligations to the user and others affected by their work. Programming assignments built around applications such as controlling the movement of a laser during eye surgery can help to address the professional, ethical and social impacts of computing. Computing faculty who are unfamiliar with the content and/or pedagogy of applied ethics are urged to take advantage of the considerable resources from ACM, IEEE-CS and other organizations.

It should be noted that the application of ethical analysis underlies every subsection of this knowledge area on Social and Professional Issues in computing. The ACM Code of Ethics and Professional Conduct - www.acm.org/about/code-of-ethics - provide guidelines that serve as the basis for the conduct of our professional work. The General Moral Imperatives provide an understanding of our commitment to personal responsibility, professional conduct, and our leadership roles.

SP. Social and Professional Practice [11 Core-Tier1 hours, 5 Core-Tier2 hours]

	Core-Tier1 hours	Core-Tier2 hours	Includes Electives
SP/Social Context	1	2	N
SP/Analytical Tools	2		N
SP/Professional Ethics	2	2	N
SP/Intellectual Property	2		Y
SP/Privacy and Civil Liberties	2		Y
SP/Professional Communication	1		Y
SP/Sustainability	1	1	Y
SP/History			Y
SP/Economies of Computing			Y
SP/Security Policies, Laws and Computer Crimes			Y

SP/Social Context

[1 Core-Tier1 hour, 2 Core-Tier2 hours]

Topics:

[Core-Tier1]

- Social implications of computing in a networked world
- Impact of social media on individualism, collectivism and culture.

[Core-Tier2]

- Growth and control of the Internet
- The digital divide (including gender, class, ethnicity, underdeveloped countries)
- Accessibility issues, including legal requirements
- Context-aware computing

Learning Outcomes:

[Core-Tier1]

1. Describe positive and negative ways in which computer technology (networks, mobile computing, cloud computing) alters modes of social interaction at the personal level. [Knowledge]
2. Identify developers' assumptions and values embedded in hardware and software design, especially as they pertain to usability for diverse populations including under-represented populations and the disabled. [Knowledge]
3. Interpret the social context of a given design and its implementation. [Knowledge]
4. Evaluate the efficacy of a given design and implementation using empirical data. [Evaluation]
5. Investigate the implications of social media on individualism versus collectivism and culture. [Application]

[Core-Tier2]

1. Discuss how Internet access serves as a liberating force for people living under oppressive forms of government; explain how limits on Internet access are used as tools of political and social repression. [Knowledge]
2. Analyze the pros and cons of reliance on computing in the implementation of democracy (e.g. delivery of social services, electronic voting). [Evaluation]
3. Describe the impact of the under-representation of diverse populations in the computing profession (e.g., industry culture, product diversity). [Knowledge]
4. Investigate the implications of context awareness in ubiquitous computing systems. [Application]

SP/Analytical Tools

[2 Core-Tier1 hours]

Topics:

[Core-Tier1]

- Ethical argumentation
- Ethical theories and decision-making
- Moral assumptions and values

Learning Outcomes:

[Core-Tier1]

1. Evaluate stakeholder positions in a given situation. [Evaluation]
2. Analyze basic logical fallacies in an argument. [Evaluation]
3. Analyze an argument to identify premises and conclusion. [Evaluation]
4. Illustrate the use of example and analogy in ethical argument. [Application]
5. Evaluate ethical tradeoffs in technical decisions. [Evaluation]

SP/Professional Ethics

[2 Core-Tier1 hours, 2 Core-Tier2 hours]

Topics:

[Core-Tier1]

- Community values and the laws by which we live
- The nature of professionalism including care, attention and discipline, fiduciary responsibility, and mentoring
- Keeping up-to-date as a professional in terms of knowledge, tools, skills, legal and professional framework as well as the ability to self-assess and computer fluency
- Codes of ethics, conduct, and practice such as the ACM/IEEE, SE, AITP, IFIP and international societies
- Accountability, responsibility and liability

[Core-Tier2]

- The role of the professional in public policy
- Maintaining awareness of consequences
- Ethical dissent and whistle-blowing
- Dealing with harassment and discrimination
- Forms of professional credentialing
- Acceptable use policies for computing in the workplace
- Ergonomics and healthy computing environments
- Time to market versus quality professional standards

Learning Outcomes:

[Core-Tier1]

1. Identify ethical issues that arise in software development and determine how to address them technically and ethically. [Knowledge]
2. Recognize the ethical responsibility of ensuring software correctness, reliability and safety. [Knowledge]
3. Describe the mechanisms that typically exist for a professional to keep up-to-date. [Knowledge]
4. Describe the strengths and weaknesses of relevant professional codes as expressions of professionalism and guides to decision-making. [Knowledge]
5. Analyze a global computing issue, observing the role of professionals and government officials in managing the problem. [Evaluation]
6. Evaluate the professional codes of ethics from the ACM, the IEEE Computer Society, and other organizations. [Evaluation]

[Core-Tier2]

1. Describe ways in which professionals may contribute to public policy. [Knowledge]
2. Describe the consequences of inappropriate professional behavior. [Knowledge]
3. Identify progressive stages in a whistle-blowing incident. [Knowledge]
4. Investigate forms of harassment and discrimination and avenues of assistance [Application]
5. Examine various forms of professional credentialing [Application]
6. Identify the social implications of ergonomic devices and the workplace environment to people's health. [Knowledge]
7. Develop a computer use policy with enforcement measures. [Evaluation]
8. Describe issues associated with industries push to focus on time to market versus enforcing quality professional standards [Knowledge]

SP/ Intellectual Property

[2 Core-Tier1 hours]

Topics:

[Core-Tier1]

- Philosophical foundations of intellectual property
- Intellectual property rights
- Intangible digital intellectual property (IDIP)
- Legal foundations for intellectual property protection
- Digital rights management
- Copyrights, patents, trademarks
- Plagiarism

[Elective]

- Foundations of the open source movement
- Software piracy

Learning Outcomes:

[Core-Tier1]

1. Discuss the philosophical bases of intellectual property. [Knowledge]
2. Discuss the rationale for the legal protection of intellectual property. [Knowledge]
3. Describe legislation aimed at digital copyright infringements. [Knowledge]
4. Critique legislation aimed at digital copyright infringements [Evaluation]
5. Identify contemporary examples of intangible digital intellectual property [Knowledge]
6. Justify uses of copyrighted materials. [Evaluation]
7. Evaluate the ethical issues inherent in various plagiarism detection mechanisms. [Evaluation]
8. Interpret the intent and implementation of software licensing. [Knowledge]
9. Discuss the issues involved in securing software patents. [Knowledge]
10. Characterize and contrast the concepts of copyright, patenting and trademarks. [Evaluation]

[Elective]

1. Identify the goals of the open source movement. [Knowledge]
2. Identify the global nature of software piracy. [Knowledge]

SP/ Privacy and Civil Liberties

[2 Core-Tier1 hours]

Topics:

[Core-Tier1]

- Philosophical foundations of privacy rights
- Legal foundations of privacy protection
- Privacy implications of widespread data collection for transactional databases, data warehouses, surveillance systems, and cloud computing
- Ramifications of differential privacy
- Technology-based solutions for privacy protection

[Elective]

- Privacy legislation in areas of practice
- Civil liberties
- Freedom of expression and its limitations

Learning Outcomes:

[Core-Tier1]

1. Discuss the philosophical basis for the legal protection of personal privacy. [Knowledge]
2. Evaluate solutions to privacy threats in transactional databases and data warehouses. [Evaluation]
3. Recognize the fundamental role of data collection in the implementation of pervasive surveillance systems (e.g., RFID, face recognition, toll collection, mobile computing). [Knowledge]
4. Recognize the ramifications of differential privacy. [Knowledge]
5. Investigate the impact of technological solutions to privacy problems. [Application]

[Elective]

1. Critique the intent, potential value and implementation of various forms of privacy legislation. [Evaluation]
2. Identify the global nature of software piracy. [Knowledge]
3. Identify strategies to enable appropriate freedom of expression. [Knowledge]

SP/ Professional Communication

[1 Core-Tier1 hour]

Topics:

[Core-Tier1]

- Reading, understanding and summarizing technical material, including source code and documentation
- Writing effective technical documentation and materials
- Dynamics of oral, written, and electronic team and group communication
- Communicating professionally with stakeholders
- Utilizing collaboration tools

[Elective]

- Dealing with cross-cultural environments
- Tradeoffs of competing risks in software projects, such as technology, structure/process, quality, people, market and financial

Learning Outcomes:

[Core-Tier1]

1. Write clear, concise, and accurate technical documents following well-defined standards for format and for including appropriate tables, figures, and references. [Application]
2. Evaluate written technical documentation to detect problems of various kinds. [Evaluation]
3. Develop and deliver a good quality formal presentation. [Evaluation]
4. Plan interactions (e.g. virtual, face-to-face, shared documents) with others in which they are able to get their point across, and are also able to listen carefully and appreciate the points of others, even when they disagree, and are able to convey to others that they have heard. [Application]
5. Describe the strengths and weaknesses of various forms of communication (e.g. virtual, face-to-face, shared documents) [Knowledge]
6. Examine appropriate measures used to communicate with stakeholders involved in a project. [Application]
7. Compare and contrast various collaboration tools. [Evaluation]

[Elective]

1. Discuss ways to influence performance and results in cross-cultural teams. [Knowledge]
2. Examine the tradeoffs and common sources of risk in software projects regarding technology, structure/process, quality, people, market and financial. [Application]

SP/ Sustainability

[1 Core-Tier1 hour, 1 Core-Tier2 hour]

Sustainability was first introduced in the CS2008 curricular guidelines. Topics in this emerging area can be naturally integrated into other knowledge areas and units.

Topics:

[Core-Tier1]

- Being a sustainable practitioner, e.g., consideration of impacts of issues, such as power consumption and resource consumption
- Explore global social and environmental impacts of computer use and disposal (e-waste)

[Core-Tier2]

- Environmental impacts of design choices in specific areas such as algorithms, operating systems, networks, databases, programming languages, or human-computer interaction
  (cross-reference: HCI/Embedded and Intelligent Systems/Energy-aware interfaces)

[Elective]

- Guidelines for sustainable design standards
- Systemic effects of complex computer-mediated phenomena (e.g. telecommuting or web shopping)
- Pervasive computing. Information processing that has been integrated into everyday objects and activities, such as smart energy systems, social networking and feedback systems to promote sustainable behavior, transportation, environmental monitoring, citizen science and activism.
- Conduct research on applications of computing to environmental issues, such as energy, pollution, resource usage, recycling and reuse, food management, farming and others.

Learning Outcomes:

[Core-Tier1]

1. Identify ways to be a sustainable practitioner [Knowledge]
2. Illustrate global social and environmental impacts of computer use and disposal (e-waste) [Application]

[Core-Tier2]

1. Describe the environmental impacts of design choices within the field of computing that relate to algorithm design, operating system design, networking design, database design, etc. [Knowledge]
2. Investigate the social and environmental impacts of new system designs through projects. [Application]

[Elective]

1. Identify guidelines for sustainable IT design or deployment [Knowledge]
2. List the sustainable effects of telecommuting or web shopping [Knowledge]
3. Investigate pervasive computing in areas such as smart energy systems, social networking, transportation, agriculture, supply-chain systems, environmental monitoring and citizen activism. [Application]
4. Develop applications of computing and assess through research areas pertaining to environmental issues (e.g. energy, pollution, resource usage, recycling and reuse, food management, farming) [Evaluation]

SP/ History

[Elective]

Topics:

- Prehistory—the world before 1946
- History of computer hardware, software, networking
- Pioneers of computing
- History of Internet

Learning Outcomes:

1. Identify significant continuing trends in the history of the computing field. [Knowledge]
2. Identify the contributions of several pioneers in the computing field. [Knowledge]
3. Discuss the historical context for several programming language paradigms. [Knowledge]
4. Compare daily life before and after the advent of personal computers and the Internet. [Evaluation]

SP/ Economies of Computing

[Elective]

Topics:

- Monopolies and their economic implications
- Effect of skilled labor supply and demand on the quality of computing products
- Pricing strategies in the computing domain
- The phenomenon of outsourcing and off-shoring; impacts on employment and on economics
- Differences in access to computing resources and the possible effects thereof
- Costing out jobs with considerations on manufacturing, hardware, software, and engineering implications
- Cost estimates versus actual costs in relation to total costs
- Entrepreneurship: prospects and pitfalls
- Use of engineering economics in dealing with finances

Learning Outcomes:

1. Summarize the rationale for antimonopoly efforts. [Knowledge]
2. Identify several ways in which the information technology industry is affected by shortages in the labor supply. [Knowledge]
3. Identify the evolution of pricing strategies for computing goods and services. [Knowledge]
4. Discuss the benefits, the drawbacks and the implications of off-shoring and outsourcing. [Knowledge]
5. Investigate and defend ways to address limitations on access to computing. [Application]

SP/ Security Policies, Laws and Computer Crimes

[Elective]

Topics:

- Examples of computer crimes and legal redress for computer criminals
- Social engineering and identity theft (cross-reference: HCI/Human Factors and Security/social engineering)
- Issues surrounding the misuse of access and breaches in security
- Motivations and ramifications of cyber terrorism and criminal hacking, “cracking”
- Effects of malware, such as viruses, worms and Trojan horses
- Crime prevention strategies
- Security policies

Learning Outcomes:

1. List examples of classic computer crimes and social engineering incidents with societal impact. [Knowledge]
2. Identify laws that apply to computer crimes [Knowledge]
3. Describe the motivation and ramifications of cyber terrorism and criminal hacking [Knowledge]
4. Examine the ethical and legal issues surrounding the misuse of access and various breaches in security [Application]
5. Discuss the professional's role in security and the trade-offs involved. [Knowledge]
6. Investigate measures that can be taken by both individuals and organizations including governments to prevent or mitigate the undesirable effects of computer crimes and identity theft [Application]
7. Write a company-wide security policy, which includes procedures for managing passwords and employee monitoring. [Application]