Computer Science Curricula 2023

Version Gamma

August 2023

The Joint Task Force on Computing Curricula

Association for Computing Machinery
(ACM)
IEEE-Computer Society
(IEEE-CS)
Association for Advancement of Artificial Intelligence
(AAAI)

Steering Committee members:

ACM members:

- Amruth N. Kumar, Ramapo College of NJ, Mahwah, NJ, USA (ACM Co-Chair)
- Monica D. Anderson, University of Alabama, Tuscaloosa, AL, USA
- Brett A. Becker, University College Dublin, Ireland
- Richard L. Blumenthal, Regis University, Denver, CO, USA
- Michael Goldweber, Denison University, Granville, OH, USA
- Pankaj Jalote, Indraprastha Institute of Information Technology, Delhi, India
- Susan Reiser, University of North Carolina Asheville, Asheville, NC, USA
- Titus Winters, Google Inc., New York, NY, USA

IEEE-CS members:

- Rajendra K. Raj, Rochester Institute of Technology (RIT), Rochester, NY, USA (IEEE-CS Co-Chair)
- Sherif G. Aly, American University in Cairo, Egypt
- Douglas Lea, SUNY Oswego, NY, USA
- Michael Oudshoorn, High Point University, High Point, NC, USA
- Marcelo Pias, Federal University of Rio Grande (FURG), Brazil
- Christian Servin, El Paso Community College, El Paso, TX, USA
- Qiao Xiang, Xiamen University, China

AAAI members:

- Eric Eaton, University of Pennsylvania, Philadelphia, PA, USA
- Susan L. Epstein, Hunter College and The Graduate Center of The City University of New York, New York, NY, USA

Table of Contents

Section 1 - Introduction

Section 2 – The Knowledge Model

Section 3 – A Competency Framework

Section 4 – Curricular Issues

Executive Summary

CS2023 is an update of the curricular guidelines last published as CS2013 by a joint task force of the ACM and IEEE Computer Society. Given the increasing importance of Artificial Intelligence, AAAI also joined forces in the CS2023 task force. These curricular guidelines address computer science, one of seven computing disciplines for which such guidelines have been issued to date.

The content of 17 of the 18 knowledge areas listed in CS2013 report has been updated. Computational Science has been dropped as a separate knowledge area (KA). Given the pervasive nature of computing applications, the role of “Society, Ethics and Professionalism” has been elevated across the curriculum. Mathematical requirements have been widened beyond Discrete Mathematics to also include Probability and Statistics. The notion of core topics has been streamlined from Tier I and Tier II in CS2013 to CS core (required topics) and KA core (recommended topics).

Given the increasing interest among educators in a competency model of the curriculum to aid in assessment, a framework has been provided for adopting institutions to create their own competency model tailored to local needs. As distinguished from knowledge model, the model used in all previous computer science curricular guidelines, competency model is offered as a complement and not a substitute for knowledge model. Adopting institutions may use either or both the models when designing their curriculum – steps have been listed for each. The knowledge model is presented in section 2 and a competency framework is presented in section 3.

Finally, the CS2023 task force engaged area experts to provide guidelines to computer science educators on social, professional, programmatic and pedagogical issues. Computer science is at a crossroads where it would be remiss to just pay lip service to these issues. These guidelines are provided in section 4.

The process used by the CS2023 task force has been collaborative (90+ KA committee members), international (all five continents), data-driven (5 large and 70 small-scale surveys) and transparent (csed.acm.org). Community engagement included numerous conference presentations and periodic postings to over a dozen Special Interest Group mailing lists. The iterative process has included at least two review and revise cycles for most knowledge areas.

Course designers may want to start with “Course packaging” section of the corresponding knowledge area in Section 2 as well as a separate section on “Course packaging” for courses that draw upon multiple knowledge areas. Curriculum designers may want to start with one of several options provided under “Curricular packaging” in Section 2. Curriculum assessment teams may want to start with the steps enumerated in Section 3 for building a competency model tailored to local needs. Program evaluators may want to use “Course packaging” and “Curricular packaging” (Section 2) and competency framework (Section 3) to compare computer science programs and facilitate credit transfers between institutions. Computer science educators and researchers may want to visit Section 4 for additional guidance on important social, professional, programmatic and pedagogical issues of the day.

Introduction

History

Several successive curricular guidelines for computer science have been published over the years as the discipline has continued to evolve:

- Curriculum 68 [1]: The first curricular guidelines were published by the Association for Computing Machinery (ACM) over 50 years ago.
- Curriculum 78 [2]: The curriculum was revised and presented in terms of core and elective courses.
- Computing Curricula 1991 [3]: The ACM teamed up with the Institute of Electrical and Electronics Engineers – Computer Society (IEEE-CS) for the first time to produce revised curricular guidelines.
- Computing Curricula 2001 [4]: For the first time, the guidelines focused only on Computer Science, with other disciplines such as computer engineering and software engineering being spun off into their own distinct curricular guidelines.
- Computer Science Curriculum 2008 [5]: This was presented as an interim revision of Computing Curricula 2001.
- Computer Science Curricula 2013 [6]: This was the most recent version of the curricula published by the ACM and IEEE-CS.

CS2023 is the next revision of computer science curricula. It is a joint effort between the ACM, IEEE-CS, and for the first time, the Association for the Advancement of Artificial Intelligence (AAAI).

All prior versions of computer science curricula focused on what is taught, referred to as a knowledge model of curricula. In such a model, related topics are grouped into a knowledge unit, and related knowledge units are grouped into a knowledge area. Computer Science Curricula 2013 [6] contained 163 knowledge units grouped into 18 knowledge areas. Learning outcomes were identified for each knowledge unit. Distinction was made between core topics that every computer science graduate must know and non-core topics that were considered to be optional.

Over the last decade, the focus of curricular design has been changing from what is taught to what is learned. What is learned is referred to as a competency model of the curriculum. Some of the early efforts to design a competency model of a curriculum were for Software Engineering (SWECOM) in 2014 [14] and Information Technology (IT2017 guidelines) [7]. These were followed by Computing Curricula CC2020 report [8] which proposed a competency model for various computing disciplines, Computer Science, Information Systems, and Data Science among them. On the heels of CC2020, competency models of curricula were produced for Information Systems 2020 [9], Associate-degree CyberSecurity [13] and Data Science 2021 [10]. The CS2023 task force set out to revise the knowledge model from the CS2013 report [6] as well as build a competency model of computer science curricula, while maintaining consistency between the two models.

It is appropriate here to acknowledge the computer science curricular work independently done by other professional bodies. These include a model curriculum for undergraduate degrees in computer science and engineering by All India Council for Technical Education in 2022 [11], and the “101 plan” of the Ministry of Education in China in 2023. Similarly, professional bodies have drafted curricular guidelines on specific areas of computer science such as parallel and distributed computing [12].

This report limits itself to computer science curricula. But, a holistic view requires consideration of the interrelatedness of computer science with other computing disciplines such as software engineering, security, and data science. For an overview of the landscape of computing education, please see the section “Computing Interrelationships” (pp 29-30) in the CC 2020 report [8].

The Task Force

The CS2023 task force consisted of a Steering Committee of 17 members and a committee for each of the 17 knowledge areas.

The Steering Committee

The ACM and IEEE-Computer Society each appointed a co-chair in January 2021 and March 2021 respectively. The rest of the Steering Committee was put together as follows:

- Three members were nominated by IEEE-CS co-chair;
- Two members were nominated by the AAAI;
- One member was nominated by ACM Committee for Computing Education in Community Colleges (CCECC);
- The remaining nine members were selected through interviews in April 2021 from among the educators who nominated themselves in response to a Call for Participation posted to multiple ACM SIG mailing lists in February 2021.

The requirements for the Steering Committee members were that they were subject experts willing to work on a volunteer basis, willing to commit to at least ten hours a month to CS2023 activities, commit to attending at least two in-person meetings a year; and were aligned with the CS2023 vision of both revising the CS2013 knowledge model and producing an appropriate competency model.

Knowledge Area Committees

In June 2021, each Steering Committee member took charge of a knowledge area, and assembled a committee of 5 – 10 subject experts, drawing members from:

- Individuals who had nominated themselves in response to the Call for Participation in February 2021;
- Industry experts; and
- Other Steering Committee members who shared interest in the knowledge area.

Knowledge Area committee members met once a month to discuss curricular revision. While the revision effort was in progress, additional subject experts who expressed interest in volunteering were added to the committees.

The Principles

The principles that have guided the work of the CS2023 task force are:

- **Collaboration:** Each knowledge area was revised by a committee of international experts from academia and industry.
- **Data-driven:** Data collected through surveys of academics and industry practitioners was used to inform the work of the task force.
- **Community outreach:** The work of the task force was continually posted and updated on the website csed.acm.org. It was presented at multiple conferences including the annual SGCSE Technical Symposium. In addition, its work was publicized through repeated postings to over a dozen ACM Special Interest Group (SIG) mailing lists.
- **Community input:** Multiple channels were provided for the community to contribute, including feedback forms and email addresses for knowledge areas and versions of the curricular guidelines.
- **Continuous review and revision:** Each version of the curricular draft was anonymously reviewed by multiple outside experts. Revision reports were produced to document how the reviews were addressed in subsequent versions of the drafts.
- **Transparency:** The work of CS2023 was documented for review and comments by the community on the website: csed.acm.org. Available information included composition of knowledge area committees, results of surveys, and the process used to form the task force.

The Process

The objectives of documenting the process are several:

- Knowledge of the process informs interpretation of the product.
- Future curricular revisions can benefit from knowledge of the process, particularly, how the process can be improved to produce better curricular guidelines.
- Curricular guidelines are a community effort. Documenting the process helps the community understand how it has contributed to the effort and how it can have a greater voice in curricular design going forward.

The overall curricular revision process was as follows:

- In 2021, surveys were conducted of the current use of CS2013 curricular guidelines and the importance of various components of curricula. The surveys were filled out by 212 academics in the United States, 191 academics from abroad and 865 industry respondents. The summaries of the surveys were incorporated into curricular revision.
- In May 2022, Version Alpha of the curriculum draft was released. It contained a revised version of the CS 2013 knowledge model. It was publicized internationally and feedback solicited. The draft of each knowledge area was sent out to reviewers suggested by the knowledge area committee. Their reviews were incorporated into the subsequence version of the curricular draft.
- In March 2023, Version Beta of the curriculum draft was released. It contained a preliminary competency model. This draft was again sent out to reviewers suggested by the knowledge area

committee as well as educators who had nominated themselves through online forms. Their reviews were incorporated into the subsequence version of the curricular draft.

- In August 2023, Version Gamma, the pre-release version of CS2023 will be posted online for a final round of comments and suggestions. It will contain course and curricular packaging information, core topics and hours, a framework for identifying atomic tasks to build a competency model and summaries of articles on curricular practices.
- The report will be released in December 2023.:

This process is illustrated in Figure 1.

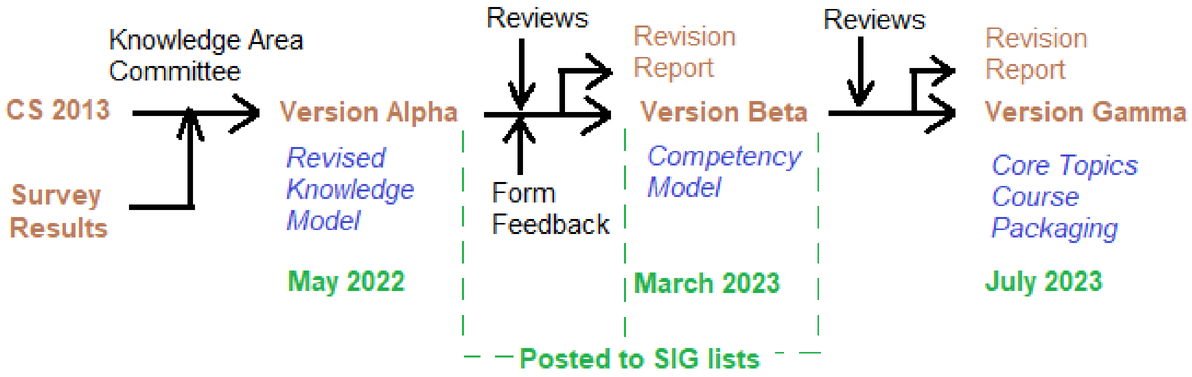


Figure 1. CS2023 Curricular revision process

Section 2

Knowledge Model

Introduction

The CS 2013 report [6] provided a knowledge model of computer science curricula. In the model, related topics were grouped together into 163 knowledge units which were in turn grouped together into 18 knowledge areas. The report listed learning outcomes for each knowledge unit and skill level for each learning outcome. It identified core topics at two levels: Tier I accounting for 165 hours of classroom instruction and Tier II accounting for 143 hours. It also included course and curriculum exemplars from several institutions of higher education.

Apart from updating the content of the CS 2013 knowledge model, several changes were made to the model, some cosmetic and others systemic, as detailed in this section.

Changes since CS2013

Knowledge Areas

Several knowledge areas were renamed, either to better emphasize their focus or to incorporate changes in the area since 2013:

- Algorithms and Complexity (AL) → Algorithmic Foundations (AL)
- Discrete Structures (DS) → Mathematical and Statistical Foundations (MSF)
- Graphics and Visualization (GV) → Graphics and Interactive Techniques (GIT)
- Information Assurance and Security (IAS) → Security (SEC)
- Information Management (IM) → Data Management (DM)
- Intelligent Systems (IS) → Artificial Intelligence (AI)
- Platform Based Development (PBD) → Specialized Platform Development (SPD)
- Programming Languages (PL) → Foundations of Programming Languages (FPL)
- Social Issues and Professional Practice (SP) → Society, Ethics and Professionalism (SEP)

Computational Science (CN) was dropped as a knowledge area because it had very little computer science content that was not also included in other knowledge areas. In CS2013, it was allocated just one core hour for modeling and simulation. Modeling was considered and rejected as an alternative to Computational Science: while applying modeling is a crosscutting theme in computer science, studying modeling for its own sake more appropriately belongs in the CS + X space.

Given that most high profile applications of computer science today draw upon statistics, the mathematical complement of computer science was expanded from Discrete Structures to also include statistics, and the knowledge area was renamed Mathematical and Statistical Foundations (MSF) to reflect this change.

Society, Ethics and Professionalism (SEP) Knowledge Unit

Given that the work of computer science graduates affects all aspects of everyday life, computer science as a discipline can no longer ignore or treat as incidental, social, ethical and professional issues. In recognition of this pervasive nature and influence of computing, effort was made to include a separate knowledge unit on Society, Ethics and Professionalism (SEP) in every other knowledge area. Topics and learning outcomes at the intersection of the knowledge area and SEP are explicitly listed in the knowledge unit to help educators call attention to these issues across the curriculum.

Topics Individually Referable

The topics in each knowledge unit and knowledge area have been enumerated for the purpose of making them individually referable. The enumeration should not be construed as implying any order or dependency among the topics. The recommended syntax for referring to a topic is:

<Knowledge area abbreviation>--<Knowledge unit abbreviation>: Decimal.alphabet.roman

For example, **AI-Search: 3.c.i** is:

3. Heuristic graph search for problem-solving
    c. Local minima and the search landscape
        i. Local vs global solutions

Skill Levels

CS2013 used the following three skill levels:

- Familiarity: “What do you know about this?”
- Usage: “What do you know how to do?”
- Assessment: “Why would you do that?”

Application is increasingly being emphasized in computer science education, especially in the context of electronic books and online courseware. So, in CS2023, “Usage” was split into “Apply” and “Develop” and four skills levels loosely aligned with revised Bloom’s taxonomy [18] were adopted as shown in Table 1.

<table>
<tr><th>Revised Bloom’s Taxonomy</th><th>Skill level with applicable verbs</th></tr>
<tr><td>Remember</td><td rowspan="2"><b>Explain:</b> define, describe, discuss, enumerate, express, identify, indicate, list, name, select, state, summarize, tabulate, translate</td></tr>
<tr><td>Understand</td></tr>
<tr><td>Apply</td><td rowspan="2"><b>Apply:</b> backup, calculate, compute, configure, debug, deploy, experiment, install, iterate, interpret, manipulate, map, measure, patch, predict, provision, randomize, recover, restore, schedule, solve, test, trace, train, virtualize</td></tr>
<tr><td rowspan="2">Analyze</td></tr>
<tr><td rowspan="2"><b>Evaluate:</b> analyze, compare, classify, contrast, distinguish, categorize, differentiate, discriminate, order, prioritize, criticize, support, decide, recommend, assess, choose, defend, predict, rank</td></tr>
<tr><td>Evaluate</td></tr>
<tr><td>Create</td><td><b>Develop:</b> combine, compile, compose, construct, create, design, generalize, integrate, modify, organize, plan, produce, rearrange, rewrite, refactor, write</td></tr>
</table>

Table 1. Skill levels and corresponding verbs and levels in revised Bloom’s taxonomy.

It should be understood that Explain is the pre-requisite skill for the other three levels. The verbs corresponding to each of the four skill levels were adopted from the work of ACM and CCECC [19].

Learning Outcomes

Learning outcomes are associated with each knowledge unit. In CS2013, skill levels were associated with each learning outcome. These skill levels were descriptive, not prescriptive, and hence, redundant. In CS2023, the learning outcomes were retained and expanded, but no longer associated with skill levels. In acknowledgment that the learning outcomes were at only one or some of the possible skills levels for each topic, learning outcomes have been renamed *Illustrative Learning Outcomes*.

Professional Dispositions

Professional dispositions are malleable values, beliefs and attitudes that enable behaviors desirable in the workplace, e.g., *persistent*, *self-directed*, etc. Whereas CS 2103 guidelines [6] emphasized the importance of dispositions in passing (Professional Practice, pp 15-16), any consideration of a competency model of the curriculum demands a more integrated treatment of dispositions.

Dispositions are generic to knowledge areas. Some dispositions are more important at certain stages in a student's development than others, e.g., *persistent* is important in introductory courses, whereas *self-directed* is important in advanced courses. *Collaborative* applies to courses with group projects whereas *meticulous* applies to mathematical foundations. So, associating dispositions with knowledge areas as opposed to individual competency statements (e.g. [16, 17]) makes it easier for the instructor to repeatedly and consistently promote dispositions during the accomplishment of tasks to which the knowledge area contributes.

In CS2023, the most relevant professional dispositions have been listed for each knowledge area. One of the sources of professional dispositions was the CC2020 report [8]. Professional dispositions serve as one of the bridges between the knowledge model (Section 2) and competency model (Section 3) of CS2023 curricular guidelines.

Core Topics

In CS2013 [6], core hours were defined along two tiers: Tier I (165 hours) and Tier II (143 hours). Computer science programs were expected to cover 100% of Tier I core topics and at least 80% of Tier II topics. While proposing this scheme, CS2013 was mindful that the number of core hours has been steadily increasing in curricular recommendations, from 280 hours in CC2001 [4] to 290 hours in CS2008 [5] and 308 hours in CS2013 [6]. Accommodating the increasing number of core hours poses a challenge for computer science programs that may want to restrict the size of the program either by design or due to necessity.

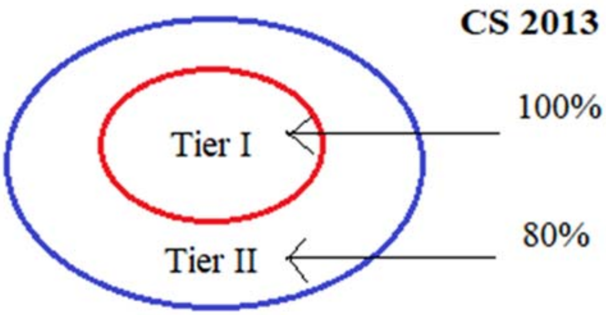


Figure 1: CS2013 Core Topics

In CS2023, a sunflower model of core topics was adopted. In it, core topics are designated as:

- CS core – topics that *every* Computer Science graduate **must** know.
- KA core – topics *recommended* for inclusion in any *dedicated* course in the knowledge area.

This model acknowledges that often, the design of curricula in smaller programs is dictated by curricular emphasis based on regional needs, the local availability of instructional expertise, and historical evolution of computer science programs. While all the programs must cover CS core topics, a program may choose to cover some knowledge areas in greater depth/breadth than other knowledge areas. In figure 2, the highlighted parts reflect the coverage of a typical computer science program. Note that the program covers some knowledge areas (KA) in great detail while others in minimal fashion or not at all.

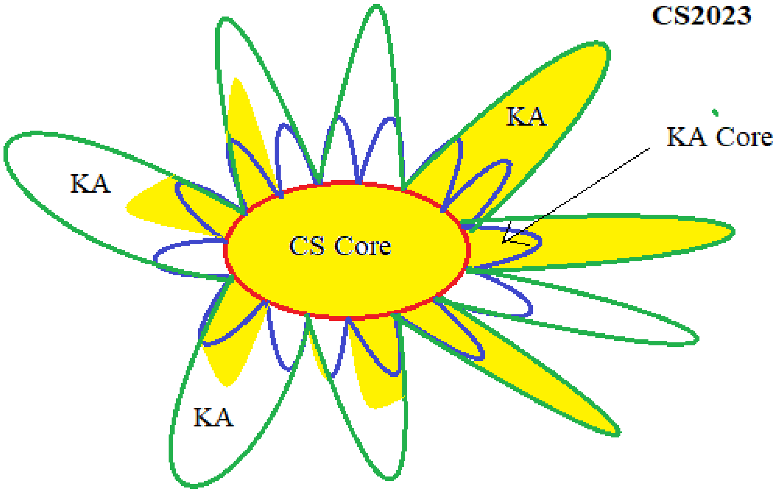


Figure 2: Sunflower model of core topics used in CS2023

The concept of KA core is fluid in CS2023 guidelines:

- Topics in Mathematical and Statistical Foundations (MSF) knowledge area are designated as KA core to indicate that they will be required for KA core topics in other knowledge areas, e.g., Statistics is needed for Machine Learning in Artificial Intelligence (AI) knowledge area. CS2023 guidelines do not purport to recommend any course packaging in Mathematical and Statistical Foundations (MSF) knowledge area other than Discrete Structures.
- Multiple distinctive courses can be carved out of some knowledge areas such as Graphics and Interactive Techniques (GIT) and Specialized Platform Development (SPD). In such cases, KA cores have been proposed for each possible course, e.g., Animation, Visualization and Image Processing arising out of Graphics and Interactive Techniques (GIT) each have their own KA cores. Their respective KA core hours are not to be considered additive for the GIT knowledge area.

Core Hours

Estimating the number of hours needed to cover core topics is a tradition of curricular guidelines. The hours are those spent in the classroom imparting knowledge regardless of the pedagogy used, and do not include the time needed for learners to develop skills or dispositions. The time needed to cover a topic in class depends on the skill-level (Explain/Apply/Evaluate/Develop) to which it is taught. So, in CS2023, skill levels were identified for each core topic in order to justify the estimation of core hours. The skill levels and hours for core topics can be found in the “Core Topics and Hours” table later in this section. The skill levels identified for core topics should be treated as recommended, not prescriptive. Table 2 shows the change in the number of core hours from CS2013 to CS2023.

Knowledge Area	**CS 2013**		**CS 2023**	
	Tier I	**Tier II**	**CS Core**	**KA Core**
Algorithmic Foundations (AL)	19	9	32	32
Architecture and Organization (AR)	0	16	9	16
Artificial Intelligence (AI)	0	10	11	12
Data Management (DM)	1	9	9	23
Foundations of Prog. Languages (FPL)	8	20	23	21
Graphics and Interactive Techniques (GIT)	2	1	4	76*
Human-Computer Interaction (HCI)	4	4	8	16
Mathematical and Statistical Foundations (MSF)	37	4	55	
Networking and Communication (NC)	3	7	7	24
Operating Systems (OS)	4	11	8	20
Parallel and Distributed Computing (PDC)	5	10	9	26
Security (SEC)	3	6	6	
Society, Ethics and Professionalism (SEP)	11	5	17	14
Software Development Fundamentals (SDF)	43	0	43	0
Software Engineering (SE)	6	22	6	23
Specialized Platform Development (SPD)	0	0	3	
Systems Fundamentals (SF)	18	9	18	9
Computational Science (CN)	1	0	Dropped	
Total	**165**	**143**	268	N/A

Table 2. Change in the number of core hours from CS2013 to CS2023.

Most knowledge areas contain a knowledge unit on Society, Ethics and Professionalism (SEP) to emphasize the pervasive importance of these issues. But, core topics and hours for SEP issues are identified only in the SEP knowledge area and not in the SEP knowledge units of other knowledge areas. This omission is meant to give educators the flexibility to decide how to cover the core SEP topics among the courses arising out of the various knowledge units.

Competency Areas

Knowledge areas, when chosen coherently, will constitute the competency area(s) of a program. Some competency areas are:

- **Software**, consisting of the knowledge areas Software Development Fundamentals (SDF), Algorithmic Foundations (AL), Foundations of Programming Languages (FPL) and Software Engineering (SE).
- **Systems**, consisting of some of the following knowledge areas: Systems Fundamentals (SF), Architecture and Organization (AR), Operating Systems (OS), Parallel and Distributed Computing (PDC), Networking and Communication (NC), Security (SEC) and Data Management (DM).

- **Applications**, consisting of some of the following knowledge areas: Graphics and Interactive Techniques (GIT), Artificial Intelligence (AI), Specialized Platform Development (SPD), Human-Computer Interaction (HCI), Security (SEC) and Data Management (DM).

Note that software competency area is in part a pre-requisite of the other two competency areas. These competency areas are another bridge between the knowledge model (Section 2) and competency model (Section 3) of CS2023 curricular guidelines.

The above list of competency areas is meant to be neither prescriptive nor comprehensive. Programs may choose to design their competency area(s) based on institutional mission and local needs. Some other competency areas that have been suggested include *Computing for the social good, scientific computing,* and *Secure computing*.

Course Packaging

A knowledge area is not a course. Many courses may be carved out of one knowledge area and one course may contain topics from multiple knowledge areas. In CS2013, course and curricular exemplars from various institutions were included in the curricular guidelines. But, adopting such exemplars from one institution to another would be affected by institutional context, including the level of preparedness of students, the availability of teaching expertise, the availability of pre- and co-requisite courses, etc. Instead, in CS2023, canonical packaging of courses has been provided in terms of knowledge areas and knowledge units, as was done in CC2001 [4].

Course packaging recommendations assume a course that meets for about 40 hours. The hours listed against each knowledge unit represent the weight suggested for the knowledge unit in a *typical* course. A course that meets for fewer or more hours may scale the hours accordingly and/or include fewer/more knowledge units in its coverage. The hours correspond to classroom coverage of the material regardless of the pedagogy used. Classroom coverage deals primarily with imparting knowledge, not the development of skills or dispositions. Course packaging instructions are suggestive, not prescriptive. They are offered as a basis for comparison of course offerings across institutions.

The Revision Process

Anonymous reviews were an integral part of the process. As summarized in Figure 1 in the Introduction to Section 1, curricular drafts went through two review and revision cycles. Table 3 lists the number of reviewers contacted and the number of reviews received for each knowledge area on its Alpha and Beta versions. For Version Beta, two numbers are listed: the number of reviewers proposed by the knowledge area committee, followed by the number of self-nominations received through online forms.

As is clear from the table, the level of community interest and involvement was not uniform across all the knowledge areas. Often, only a small fraction of the contacted reviewers returned reviews. The number of self-nominations also varied across knowledge areas. Nevertheless, knowledge area committees completed the review loop by posting a revision report after each review cycle. The reports are accessible from the respective knowledge area pages on the website csed.acm.org.

Knowledge Area	**Version Alpha**		**Version Beta**	
	Contacted	**Reviewed**	**Contacted***	**Reviewed**
Algorithmic Foundations (AL)	6	3	9/5	5
Architecture and Organization (AR)	15	1	/3	2

Artificial Intelligence (AI)	1		10	
Data Management (DM)	10	2	/2	2
Foundations of Prog. Languages (FPL)	10	3	16/4	4
Graphics and Interactive Techniques (GIT)	3	3	3	3
Human-Computer Interaction (HCI)	9	3	15/2	2
Networking and Communication (NC)	9	1	10/3	4
Operating Systems (OS)	7	3	6/1	2
Parallel and Distributed Computing (PDC)	4	4	/1	1
Security			6/2	3
Society, Ethics and Professionalism (SEP)	8		7/1	3
Software Development Fundamentals (SDF)	4	2	10/2	8
Software Engineering (SE)	4	3	10/1	4
Specialized Platform Development (SPD)	9	3	8	
Systems Fundamentals (SF)	5	1	5/2	2

Table 3. Summary of review and revision process.

The process for determining core topics and hours was as follows:

1. CS2013 Tier I core topics were converted to CS core topics, and Tier II core topics into KA core topics;
2. Topics were moved among CS core, KA core and non-core by the knowledge area committee as appropriate;
3. Skill levels were identified for each CS and KA core topic in order to justify the core hours dedicated to the topic;
4. Core topics shared between knowledge areas were identified so that their hours would be counted only once;
5. 70 CS core surveys were conducted wherein educators were asked whether topics identified as CS core should remain in CS core;
6. Based on the results of the surveys filled out by 198 educators, CS core topics were whittled down.

Mathematical and Statistical Foundations topics were determined using multiple inputs: from the mathematical requirements identified in other knowledge areas, from the computer science theory community, from various reports (example: the Park City report on data science) and, critically, two surveys, one distributed to faculty and one to industry practitioners. The first survey was issued to computer science faculty (with nearly 600 faculty responding) across a variety of institutional types and in various countries to obtain a snapshot of current practices in mathematical foundations and to solicit opinion on the importance of particular topics beyond the traditional discrete mathematics. The second survey was sent to industry employees (approximately 680 respondents) requesting their views on curricular topics and components.

How to adopt/adapt the knowledge model

1. Identify the competency area(s) targeted by the curriculum based on local needs;

2. Based on the selected competency area(s), select the knowledge areas of coverage while taking into account availability of instructional expertise and coverage of CS core topics;

3. For each knowledge area identified in step 2, start with one or more course packaging suggestions. For each course:

a. Add/subtract/scale knowledge units as appropriate;

b. Ensure that all CS core topics remain included;

c. Maximize the KA core topics recommended for the knowledge area;

d. Eliminate duplication of topics shared with other courses in the curriculum;

Body of Knowledge

	Knowledge Area	**Knowledge Units**	**CS Core**	**KA Core**
AI	Artificial Intelligence	12	11	13
AL	Algorithmic Foundations	5	32	32
AR	Architecture and Organization	9	9	16
DM	Data Management	12	9	21
FPL	Foundations of Programming Languages	20	23	20
GIT	Graphics and Interactive Techniques	11	4	
HCI	Human-Computer Interaction	6	8	16
MSF	Mathematical and Statistical Foundations	5	55	200
NC	Networking and Communication	8	7	24
OS	Operating Systems	14	8	17
PDC	Parallel and Distributed Computing	5	9	26
SDF	Software Development Fundamentals	5	43	0
SE	Software Engineering	9	6	21
SEC	Security	6	6	33
SEP	Society, Ethics and Professionalism	11	17	14
SF	Systems Fundamentals	8	18	9
SPD	Specialized Platform Development	8	3	
	Total		268	N/A

Artificial Intelligence (AI)

Preamble

Artificial intelligence (AI) studies problems that are difficult or impractical to solve with traditional algorithmic approaches. These problems are often reminiscent of those considered to require human intelligence, and the resulting AI solution strategies typically generalize over classes of problems. AI techniques are now pervasive in computing, supporting everyday applications such as email, social media, photography, financial markets, and intelligent virtual assistants (e.g., Siri, Alexa). These techniques are also used in the design and analysis of autonomous agents that perceive their environment and interact rationally with it, such as self-driving vehicles and other robots.

Traditionally, AI has included a mix of symbolic and subsymbolic approaches. The solutions it provides rely on a broad set of general and specialized knowledge representation schemes, problem solving mechanisms, and optimization techniques. These approaches deal with perception (e.g., speech recognition, natural language understanding, computer vision), problem solving (e.g., search, planning, optimization), acting (e.g., robotics, task-automation, control), and the architectures needed to support them (e.g., single agents, multi-agent systems). Machine learning may be used within each of these aspects, and can even be employed end-to-end across all of them. The study of Artificial Intelligence prepares students to determine when an AI approach is appropriate for a given problem, identify appropriate representations and reasoning mechanisms, implement them, and evaluate them with respect to both performance and their broader societal impact.

Over the past decade, the term “artificial intelligence” has become commonplace within businesses, news articles, and everyday conversation, driven largely by a series of high-impact machine learning applications. These advances were made possible by the widespread availability of large datasets, increased computational power, and algorithmic improvements. In particular, there has been a shift from engineered representations to representations learned automatically through optimization over large datasets. The resulting advances have put such terms as “neural networks” and “deep learning” into everyday vernacular. Businesses now advertise AI-based solutions as value-additions to their services, so that “artificial intelligence” is now both a technical term and a marketing keyword. Other disciplines, such as biology, art, architecture, and finance, increasingly use AI techniques to solve problems within their disciplines.

For the first time in our history, the broader population has access to sophisticated AI-driven tools, including tools to generate essays or poems from a prompt, photographs or artwork from a description, and fake photographs or videos depicting real people. AI technology is now in widespread use in stock trading, curating our news and social media feeds, automated evaluation of job applicants, detection of medical conditions, and influencing prison sentencing through recidivism prediction. Consequently, AI technology can have significant societal impacts that must be understood and considered when developing and applying it.

Changes since CS 2013

To reflect this recent growth and societal impact, the knowledge area has been revised from CS 2013 in the following ways:

- The name has changed from "Intelligent Systems" to "Artificial Intelligence," to reflect the most common terminology used for these topics within the field and its more widespread use outside the field.
- An increased emphasis on neural networks and representation learning reflects the recent advances in the field. Given its key role throughout AI, search is still emphasized but there is a slight reduction on symbolic methods in favor of understanding subsymbolic methods and learned representations. It is important, however, to retain knowledge-based and symbolic approaches within the AI curriculum because these methods offer unique capabilities, are used in practice, ensure a broad education, and because more recent neurosymbolic approaches integrate both learned and symbolic representations.
- There is an increased emphasis on practical applications of AI, including a variety of areas (e.g., medicine, sustainability, social media, etc.). This includes explicit discussion of tools that employ deep generative models (e.g., ChatGPT, DALL-E, Midjourney) and are now in widespread use, covering how they work at a high level, their uses, and their shortcomings/pitfalls.
- The curriculum reflects the importance of understanding and assessing the broader societal impacts and implications of AI methods and applications, including issues in AI ethics, fairness, trust, and explainability.
- The AI knowledge area includes connections to data science through cross-connections with the Data Management knowledge area.
- There are explicit goals to develop basic AI literacy and critical thinking in every computer science student, given the breadth of interconnections between AI and other knowledge areas in practice.

Note: Consider recent AI advances when using this curriculum

The field of AI is undergoing rapid development and increasingly widespread applications. Since the first draft of this document, several new techniques (e.g., generative networks, large language models) have become widely used and so were added to the CS or KA cores. This document is as current as we can make it in 2023. However, we expect such rapid changes to continue in the subfield of AI during the expected ten-year life of this document. Consequently, it is imperative that faculty teaching AI understand current advances and consider whether these advances should be taught in order to keep the curriculum current.

Core Hours

Knowledge Units	CS Core	KA Core
Fundamental Issues	2	1
Search	2 + 3 (AL) †	4

Fundamental Knowledge Representation and Reasoning	1 + 1 (MSF) [‡]	2
Machine Learning	4	4
Applications and Societal Impact	2	2
Probabilistic Representation and Reasoning		
Planning		
Logical Representation and Reasoning		
Agents		
Natural Language Processing		
Robotics		
Perception and Computer Vision		
Total	**11**	**13**

[†] 5 CS Core hours, 3 of which are counted under AL Algorithms (Uninformed search)
[‡] 2 CS Core hours, 1 of which is counted under MSF (Probability)

Knowledge Units

AI-Introduction: Fundamental Issues

CS Core:

1. Overview of AI problems, Examples of successful recent AI applications
2. Definitions of agents with examples (e.g., reactive, deliberative)
3. What is intelligent behavior?
    a. The Turing test and its flaws
    b. Multimodal input and output
    c. Simulation of intelligent behavior
    d. Rational versus non-rational reasoning
4. Problem characteristics
    a. Fully versus partially observable
    b. Single versus multi-agent
    c. Deterministic versus stochastic
    d. Static versus dynamic
    e. Discrete versus continuous

5. Nature of agents
    a. Autonomous, semi-autonomous, mixed-initiative autonomy
    b. Reflexive, goal-based, and utility-based
    c. Decision making under uncertainty and with incomplete information
    d. The importance of perception and environmental interactions
    e. Learning-based agents
    f. Embodied agents
        i. sensors, dynamics, effectors
6. AI Applications, growth, and Impact (economic, societal, ethics)

KA Core:

7. Practice identifying problem characteristics in example environments
8. Additional depth on nature of agents with examples
9. Additional depth on AI Applications, growth, and Impact (economic, societal, ethics)

Non-Core:

10. Philosophical issues
11. History of AI

Illustrative Learning Outcomes:

1. Describe the Turing test and the “Chinese Room” thought experiment.
2. Differentiate between optimal reasoning/behavior and human-like reasoning/behavior.
3. Determine the characteristics of a specific problem.

AI-Search: Search

CS Core:

- State space representation of a problem
    a. Specifying states, goals, and operators
    b. Factoring states into representations (hypothesis spaces)
    c. Problem solving by graph search
        i. e.g., Graphs as a space, and tree traversals as exploration of that space
        ii. Dynamic construction of the graph (you’re not given it upfront)
- Uninformed graph search for problem solving (See also: AL-Fundamentals:12)
    a. Breadth-first search
    b. Depth-first search
        i. With iterative deepening
    c. Uniform cost search
- Heuristic graph search for problem solving (See also: AL-Strategies)
    a. Heuristic construction and admissibility
    b. Hill-climbing
    c. Local minima and the search landscape
        i. Local vs global solutions
    d. Greedy best-first search

    e. A* search
- Space and time complexities of graph search algorithms

KA Core:

- Bidirectional search
- Beam search
- Two-player adversarial games
    a. Minimax search
    b. Alpha-beta pruning
        i. Ply cutoff
- Implementation of A* search

Non-Core:

- Understanding the search space
    a. Constructing search trees
    b. Dynamic search spaces
    c. Combinatorial explosion of search space
    d. Search space topology (ridges, saddle points, local minima, etc.)
- Local search
- Constraint satisfaction
- Tabu search
- Variations on A* (IDA*, SMA*, RBFS)
- Two-player adversarial games
    a. The horizon effect
    b. Opening playbooks / endgame solutions
    c. What it means to “solve” a game (e.g., checkers)
- Implementation of minimax search, beam search
- Expectimax search (MDP-solving) and chance nodes
- Stochastic search
    . Simulated annealing
    a. Genetic algorithms
    b. Monte-Carlo tree search

Illustrative Learning Outcomes:

1. Design the state space representation for a puzzle (e.g., N-queens or 3-jug problem)
2. Select and implement an appropriate uninformed search algorithm for a problem (e.g., tic-tac-toe), and characterize its time and space complexities.
3. Select and implement an appropriate informed search algorithm for a problem after designing a helpful heuristic function (e.g., a robot navigating a 2D gridworld).
4. Evaluate whether a heuristic for a given problem is admissible/can guarantee an optimal solution.
5. Apply minimax search in a two-player adversarial game (e.g., connect four), using heuristic evaluation at a particular depth to compute the scores to back up. [KA core]
6. Design and implement a genetic algorithm solution to a problem.

7. Design and implement a simulated annealing schedule to avoid local minima in a problem.
8. Design and implement A*/beam search to solve a problem, and compare it against other search algorithms in terms of the solution cost, number of nodes expanded, etc.
9. Apply minimax search with alpha-beta pruning to prune search space in a two-player adversarial game (e.g., connect four).
10. Compare and contrast genetic algorithms with classic search techniques, explaining when it is most appropriate to use a genetic algorithm to learn a model versus other forms of optimization (e.g., gradient descent).
11. Compare and contrast various heuristic searches vis-a-vis applicability to a given problem.

AI-KRR: Fundamental Knowledge Representation and Reasoning

CS Core:

1. Types of representations
    a. Symbolic, logical
        i. Creating a representation from a natural language problem statement
    b. Learned subsymbolic representations
    c. Graphical models (e.g., naive Bayes, Bayesian network)
2. Review of probabilistic reasoning, Bayes theorem (See also: MSF-TODO)
3. Bayesian reasoning
    a. Bayesian inference

KA Core:

4. Random variables and probability distributions
    a. Axioms of probability
    b. Probabilistic inference
    c. Bayes' Rule (derivation)
    d. Bayesian inference (more complex examples)
5. Independence
6. Conditional Independence
7. Markov chains and Markov models
8. Utility and decision making

Illustrative Learning Outcomes:

1. Given a natural language problem statement, encode it as a symbolic or logical representation.
2. Explain how we can make decisions under uncertainty, using concepts such as Bayes theorem and utility.
3. Make a probabilistic inference in a real-world problem using Bayes' theorem to determine the probability of a hypothesis given evidence.
4. Apply Bayes' rule to determine the probability of a hypothesis given evidence.
5. Compute the probability of outcomes and test whether outcomes are independent.

AI-ML: Machine Learning

CS Core:

1. Definition and examples of a broad variety of machine learning tasks
    a. Supervised learning
        i. Classification
        ii. Regression
    b. Reinforcement learning
    c. Unsupervised learning
        i. Clustering
2. Fundamental ideas:
    a. No free lunch: no one learner can solve all problems; representational design decisions have consequences
    b. sources of error and undecidability in machine learning
3. A simple statistical-based supervised learning such as linear regression or decision trees
    a. Focus on how they work without going into mathematical or optimization details; enough to understand and use existing implementations correctly
4. The overfitting problem / controlling solution complexity (regularization, pruning – intuition only)
    a. The bias (underfitting) - variance (overfitting) tradeoff
5. Working with Data
    a. Data preprocessing
        i. Importance and pitfalls of
    b. Handling missing values (imputing, flag-as-missing)
        i. Implications of imputing vs flag-as-missing
    c. Encoding categorical variables, encoding real-valued data
    d. Normalization/standardization
    e. Emphasis on real data, not textbook examples
6. Representations
    a. Hypothesis spaces and complexity
    b. Simple basis feature expansion, such as squaring univariate features
    c. Learned feature representations
7. Machine learning evaluation
    a. Separation of train, validation, and test sets
    b. Performance metrics for classifiers
    c. Estimation of test performance on held-out data
    d. Tuning the parameters of a machine learning model with a validation set
    e. Importance of understanding what your model is actually doing, where its pitfalls/shortcomings are, and the implications of its decisions
8. Basic neural networks
    a. Fundamentals of understanding how neural networks work and their training process, without details of the calculations
    b. Basic introduction to generative neural networks (large language models, etc.)
9. Ethics for Machine Learning (See also: SEP-Context)
    a. Focus on real data, real scenarios, and case studies.
    b. Dataset/algorithmic/evaluation bias

KA Core:

10. Formulation of simple machine learning as an optimization problem, such as least squares linear regression or logistic regression
    a. Objective function
    b. Gradient descent
    c. Regularization to avoid overfitting (mathematical formulation)
11. Ensembles of models
    a. Simple weighted majority combination
12. Deep learning
    a. Deep feed-forward networks (intuition only, no math)
    b. Convolutional neural networks (intuition only, no math)
    c. Visualization of learned feature representations from deep nets
    d. Other architectures (generative NN, recurrent NN, transformers, etc.)
13. Performance evaluation
    a. Other metrics for classification (e.g., error, precision, recall)
    b. Performance metrics for regressors
    c. Confusion matrix
    d. Cross-validation
        i. Parameter tuning (grid/random search, via cross-validation)
14. Overview of reinforcement learning
15. Two or more applications of machine learning algorithms
    a. E.g., medicine and health, economics, vision, natural language, robotics, game play
16. Ethics for Machine Learning
    a. Continued focus on real data, real scenarios, and case studies (See also: SEP-Context)
    b. Privacy (See also: SEP-Privacy)
    c. Fairness (See also: SEP-Privacy)

Non-Core:

17. General statistical-based learning, parameter estimation (maximum likelihood)
18. Supervised learning
    a. Decision trees
    b. Nearest-neighbor classification and regression
    c. Learning simple neural networks / multi-layer perceptrons
    d. Linear regression
    e. Logistic regression
    f. Support vector machines (SVMs) and kernels
    g. Gaussian Processes
19. Overfitting
    a. The curse of dimensionality
    b. Regularization (math computations, L_2 and L_1 regularization)
20. Experimental design
    a. Data preparation (e.g., standardization, representation, one-hot encoding)
    b. Hypothesis space

c. Biases (e.g., algorithmic, search)
d. Partitioning data: stratification, training set, validation set, test set
e. Parameter tuning (grid/random search, via cross-validation)
f. Performance evaluation
i. Cross-validation
ii. Metric: error, precision, recall, confusion matrix
iii. Receiver operating characteristic (ROC) curve and area under ROC curve

21. Bayesian learning (Cross-Reference AI/Reasoning Under Uncertainty)
    a. Naive Bayes and its relationship to linear models
    b. Bayesian networks
    c. Prior/posterior
    d. Generative models
22. Deep learning
    a. Deep feed-forward networks
    b. Neural tangent kernel and understanding neural network training
    c. Convolutional neural networks
    d. Autoencoders
    e. Recurrent networks
    f. Representations and knowledge transfer
    g. Adversarial training and generative adversarial networks
23. Representations
    a. Manually crafted representations
    b. Basis expansion
    c. Learned representations (e.g., deep neural networks)
24. Unsupervised learning and clustering
    a. K-means
    b. Gaussian mixture models
    c. Expectation maximization (EM)
    d. Self-organizing maps
25. Graph analysis (e.g., PageRank)
26. Semi-supervised learning
27. Graphical models (See also: AI/Probabilistic Representation and Reasoning)
28. Ensembles
    a. Weighted majority
    b. Boosting/bagging
    c. Random forest
    d. Gated ensemble
29. Learning theory
    a. General overview of learning theory / why learning works
    b. VC dimension
    c. Generalization bounds
30. Reinforcement learning
    a. Exploration vs. exploitation trade-off
    b. Markov decision processes

    c. Value and policy iteration
    d. Policy gradient methods
    e. Deep reinforcement learning

31. Explainable / interpretable machine learning
    a. Understanding feature importance (e.g., LIME, Shapley values)
    b. Interpretable models and representations
32. Recommender systems
33. Hardware for machine learning
    a. GPUs / TPUs
34. Application of machine learning algorithms to:
    a. Medicine and health
    b. Economics
    c. Education
    d. Vision
    e. Natural language
    f. Robotics
    g. Game play
    h. Data mining (Cross-reference IM/Data Mining)
35. Ethics for Machine Learning
    a. Continued focus on real data, real scenarios, and case studies (See also: SEP-Context)
    b. In depth exploration of dataset/algorithmic/evaluation bias, data privacy, and fairness (See also: SEP-Privacy, SEP-Context)
    c. Trust / explainability

Illustrative Learning Outcomes:

1. Describe the differences among the three main styles of learning: supervised, reinforcement, and unsupervised.
2. Differentiate the terms of AI, machine learning, and deep learning.
3. Frame an application as a classification problem, including the available input features and output to be predicted (e.g., identifying alphabetic characters from pixel grid input).
4. Apply two or more simple statistical learning algorithms (such as k-nearest-neighbors and logistic regression) to a classification task and measure the classifiers' accuracy.
5. Identify overfitting in the context of a problem and learning curves and describe solutions to overfitting.
6. Explain how machine learning works as an optimization/search process.
7. Implement a statistical learning algorithm and the corresponding optimization process to train the classifier and obtain a prediction on new data.
8. Describe the neural network training process and resulting learned representations
9. Explain proper ML evaluation procedures, including the differences between training and testing performance, and what can go wrong with the evaluation process leading to inaccurate reporting of ML performance.
10. Compare two machine learning algorithms on a dataset, implementing the data preprocessing and evaluation methodology (e.g., metrics and handling of train/test splits) from scratch.

11. Visualize the training progress of a neural network through learning curves in a well-established toolkit (e.g., TensorBoard) and visualize the learned features of the network.
12. Implement simple algorithms for supervised learning, reinforcement learning, and unsupervised learning.
13. Determine which of the three learning styles is appropriate to a particular problem domain.
14. Compare and contrast each of the following techniques, providing examples of when each strategy is superior: decision trees, logistic regression, naive Bayes, neural networks, and belief networks.
15. Evaluate the performance of a simple learning system on a real-world dataset.
16. Characterize the state of the art in learning theory, including its achievements and its shortcomings.
17. Explain the problem of overfitting, along with techniques for detecting and managing the problem.
18. Explain the triple tradeoff among the size of a hypothesis space, the size of the training set, and performance accuracy.

AI-SEP: Applications and Societal Impact

Note: There is substantial benefit to studying applications and ethics/fairness/trust/explainability in a curriculum alongside the methods and theory that it applies to, rather than covering ethics in a separate, dedicated class session. Whenever possible, study of these topics should be integrated alongside other modules, such as exploring how decision trees could be applied to a specific problem in environmental sustainability such as land use allocation, then assessing the social/environmental/ethical implications of doing so.

CS/KA Core: *For the CS core, cover at least one application and an overview of the societal issues of AI/ML. The KA core should go more in-depth with one or more additional applications, more in-depth on deep generative models, and an analysis and discussion of the social issues.*

1. Applications of AI to a broad set of problems and diverse fields, such as medicine, health, sustainability, social media, economics, education, robotics, etc. (choose one for CS Core, at least one additional for KA core)
    a. Formulating and evaluating a specific application as an AI problem
    b. Data availability and cleanliness
        i. Basic data cleaning and preprocessing
        ii. Data set bias
    c. Algorithmic bias
    d. Evaluation bias
2. Deployed deep generative models
    a. High-level overview of deep image generative models (e.g. as of 2023, DALL-E, Midjourney, Stable Diffusion, etc.), how they work, their uses, and their shortcomings/pitfalls.
    b. High-level overview of large language models (e.g. as of 2023, ChatGPT, Bard, etc.), how they work, their uses, and their shortcomings/pitfalls.
3. Societal impact of AI

  a. Ethics
  b. Fairness
  c. Trust / explainability
  d. Privacy and usage of training data
  e. Human autonomy and oversight
  f. Sustainability

Illustrative Learning Outcomes:

1. Given a real-world application domain and problem, formulate an AI solution to it, identifying proper data/input, preprocessing, representations, AI techniques, and evaluation metrics/methodology.
2. Analyze the societal impact of one or more specific real-world AI applications, identifying issues regarding ethics, fairness, bias, trust, and explainability.
3. Describe some of the failure modes of current deep generative models for language or images, and how this could affect their use in an application.

AI-LRR: Logical Representation and Reasoning

Non-Core:

1. Review of propositional and predicate logic (see also: DS/Basic Logic)
2. Resolution and theorem proving (propositional logic only)
   a. Forward chaining, backward chaining
3. Knowledge representation issues
   a. Description logics
   b. Ontology engineering
4. Semantic web
5. Non-monotonic reasoning (e.g., non-classical logics, default reasoning)
6. Argumentation
7. Reasoning about action and change (e.g., situation and event calculus)
8. Temporal and spatial reasoning
9. Logic programming
   a. Prolog, Answer Set Programming
10. Rule-based Expert Systems
11. Semantic networks
12. Model-based and Case-based reasoning

Illustrative Learning Outcomes:

1. Translate a natural language (e.g., English) sentence into a predicate logic statement.
2. Convert a logic statement into clausal form.
3. Apply resolution to a set of logic statements to answer a query.
4. Compare and contrast the most common models used for structured knowledge representation, highlighting their strengths and weaknesses.
5. Identify the components of non-monotonic reasoning and its usefulness as a representational mechanism for belief systems.

6. Compare and contrast the basic techniques for representing uncertainty.
7. Compare and contrast the basic techniques for qualitative representation.
8. Apply situation and event calculus to problems of action and change.
9. Explain the distinction between temporal and spatial reasoning, and how they interrelate.
10. Explain the difference between rule-based, case-based and model-based reasoning techniques.
11. Define the concept of a planning system and how it differs from classical search techniques.
12. Describe the differences between planning as search, operator-based planning, and propositional planning, providing examples of domains where each is most applicable.
13. Explain the distinction between monotonic and non-monotonic inference.

AI-Prob: Probabilistic Representation and Reasoning

Non-Core:

1. Conditional Independence review
2. Knowledge representations
    a. Bayesian Networks
        i. Exact inference and its complexity
        ii. Markov blankets and d-separation
        iii. Randomized sampling (Monte Carlo) methods (e.g. Gibbs sampling)
    b. Markov Networks
    c. Relational probability models
    d. Hidden Markov Models
3. Decision Theory
    a. Preferences and utility functions
    b. Maximizing expected utility
    c. Game theory

Illustrative Learning Outcomes:

1. Compute the probability of a hypothesis given the evidence in a Bayesian network.
2. Explain how conditional independence assertions allow for greater efficiency of probabilistic systems.
3. Identify examples of knowledge representations for reasoning under uncertainty.
4. State the complexity of exact inference. Identify methods for approximate inference.
5. Design and implement at least one knowledge representation for reasoning under uncertainty.
6. Describe the complexities of temporal probabilistic reasoning.
7. Design and implement an HMM as one example of a temporal probabilistic system.
8. Describe the relationship between preferences and utility functions.
9. Explain how utility functions and probabilistic reasoning can be combined to make rational decisions.

AI-Planning: Planning

Non-Core:

1. Review of propositional and first-order logic
2. Planning operators and state representations
3. Total order planning
4. Partial-order planning
5. Plan graphs and GraphPlan
6. Hierarchical planning
7. Planning languages and representations
    a. PDDL
8. Multi-agent planning
9. MDP-based planning
10. Interconnecting planning, execution, and dynamic replanning
    a. Conditional planning
    b. Continuous planning
    c. Probabilistic planning

Illustrative Learning Outcomes:

1. Construct the state representation, goal, and operators for a given planning problem.
2. Encode a planning problem in PDDL and use a planner to solve it.
3. Given a set of operators, initial state, and goal state, draw the partial-order planning graph and include ordering constraints to resolve all conflicts
4. Construct the complete planning graph for GraphPlan to solve a given problem.

AI-Agents: Agents

(Cross-reference HCI/Collaboration and Communication)

Non-Core:

1. Agent architectures (e.g., reactive, layered, cognitive)
2. Agent theory (including mathematical formalisms)
3. Rationality, Game Theory
    . Decision-theoretic agents
    a. Markov decision processes (MDP)
    b. Bandit algorithms
4. Software agents, personal assistants, and information access
    a. Collaborative agents
    b. Information-gathering agents
    c. Believable agents (synthetic characters, modeling emotions in agents)
5. Learning agents
6. Cognitive architectures (e.g., ACT-R, SOAR, ICARUS, FORR)
    a. Capabilities (perception, decision making, prediction, knowledge maintenance, etc.)
    b. Knowledge representation, organization, utilization, acquisition, and refinement
    c. Applications and evaluation of cognitive architectures
7. Multi-agent systems
    a. Collaborating agents
    b. Agent teams

    c. Competitive agents (e.g., auctions, voting)
    d. Swarm systems and biologically inspired models
    e. Multi-agent learning
8. Human-agent interaction
    a. Communication methodologies (verbal and non-verbal)
    b. Practical issues
    c. Applications
        i. Trading agents, supply chain management

Illustrative Learning Outcomes:

1. Characterize and contrast the standard agent architectures.
2. Describe the applications of agent theory to domains such as software agents, personal assistants, and believable agents.
3. Describe the primary paradigms used by learning agents.
4. Demonstrate using appropriate examples how multi-agent systems support agent interaction.
5. Construct an intelligent agent using a well-established cognitive architecture (ACT-R, SOAR) for solving a specific problem.

AI-NLP: Natural Language Processing

Non-Core:

1. Deterministic and stochastic grammars
2. Parsing algorithms
    a. CFGs and chart parsers (e.g. CYK)
    b. Probabilistic CFGs and weighted CYK
3. Representing meaning / Semantics
    a. Logic-based knowledge representations
    b. Semantic roles
    c. Temporal representations
    d. Beliefs, desires, and intentions
4. Corpus-based methods
5. N-grams and HMMs
6. Smoothing and backoff
7. Examples of use: POS tagging and morphology
8. Information retrieval (See also: IM/Information Storage and Retrieval)
    a. Vector space model
        i. TF & IDF
    b. Precision and recall
9. Information extraction
10. Language translation
11. Text classification, categorization
    a. Bag of words model
12. Deep learning for NLP (See also: AI/Machine Learning)
    a. RNNs

   b. Transformers
   c. Multi-modal embeddings (e.g., images + text)
   d. Generative language models

Illustrative Learning Outcomes:

1. Define and contrast deterministic and stochastic grammars, providing examples to show the adequacy of each.
2. Simulate, apply, or implement classic and stochastic algorithms for parsing natural language.
3. Identify the challenges of representing meaning.
4. List the advantages of using standard corpora. Identify examples of current corpora for a variety of NLP tasks.
5. Identify techniques for information retrieval, language translation, and text classification.
6. Implement a TF/IDF transform, use it to extract features from a corpus, and train an off-the-shelf machine learning algorithm using those features to do text classification.

AI-Robo: Robotics

(See also: SPD/Robot Platforms)

Non-Core:

1. Overview: problems and progress
   a. State-of-the-art robot systems, including their sensors and an overview of their sensor processing
   b. Robot control architectures, e.g., deliberative vs. reactive control and Braitenberg vehicles
   c. World modeling and world models
   d. Inherent uncertainty in sensing and in control
2. Sensors and effectors
   a. Sensors: LIDAR, sonar, vision, depth, stereoscopic, event cameras, microphones, haptics, etc.
   b. Effectors: wheels, arms, grippers, etc.
3. Coordinate frames, translation, and rotation (2D and 3D)
4. Configuration space and environmental maps
5. Interpreting uncertain sensor data
6. Localization and mapping
7. Navigation and control
8. Forward and inverse kinematics
9. Motion path planning and trajectory optimization
10. Joint control and dynamics
11. Vision-based control
12. Multiple-robot coordination and collaboration
13. Human-robot interaction (See also: HCI)
    a. Shared workspaces
    b. Human-robot teaming and physical HRI
    c. Social assistive robots

   d. Motion/task/goal prediction
   e. Collaboration and communication (explicit vs implicit, verbal or symbolic vs non-verbal or visual)
   f. Trust

Illustrative Learning Outcomes:

(Note: Due to the expense of robot hardware, all of these could be done in simulation or with low-cost educational robotic platforms.)

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.
2. Integrate sensors, actuators, and software into a robot designed to undertake some task.
3. Program a robot to accomplish simple tasks using deliberative, reactive, and/or hybrid control architectures.
4. Implement fundamental motion planning algorithms within a robot configuration space.
5. Characterize the uncertainties associated with common robot sensors and actuators; articulate strategies for mitigating these uncertainties.
6. List the differences among robots' representations of their external environment, including their strengths and shortcomings.
7. Compare and contrast at least three strategies for robot navigation within known and/or unknown environments, including their strengths and shortcomings.
8. Describe at least one approach for coordinating the actions and sensing of several robots to accomplish a single task.
9. Compare and contrast a multi-robot coordination and a human-robot collaboration approach, and attribute their differences to differences between the problem settings.

AI-Vision: Perception and Computer Vision

Non-Core:

1. Computer vision
   a. Image acquisition, representation, processing and properties
   b. Shape representation, object recognition, and segmentation
   c. Motion analysis
   d. Generative models
2. Audio and speech recognition
3. Touch and proprioception
4. Other modalities (e.g., olfaction)
5. Modularity in recognition
6. Approaches to pattern recognition. (See also: AI/Machine Learning)
   a. Classification algorithms and measures of classification quality
   b. Statistical techniques
   c. Deep learning techniques

Illustrative Learning Outcomes:

1. Summarize the importance of image and object recognition in AI and indicate several significant applications of this technology.

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.
3. Implement 2d object recognition based on contour- and/or region-based shape representations.
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.
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.
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.
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.
8. Implement a classification algorithm that segments input percepts into output categories and quantitatively evaluates the resulting classification.
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.
10. Describe at least three classification approaches, their prerequisites for applicability, their strengths, and their shortcomings.
11. Implement and evaluate a deep learning solution to problems in computer vision, such as object or scene recognition.

Professional Dispositions

- **Meticulousness:** Attention must be paid to details when implementing AI and machine learning algorithms, requiring students to be meticulous to detail.
- **Persistence:** AI techniques often operate in partially observable environments and optimization processes may have cascading errors from multiple iterations. Getting AI techniques to work predictably takes trial and error, and repeated effort. These call for persistence on the part of the student.
- **Responsible:** Applications of AI can have significant impacts on society, affecting both individuals and large populations. This calls for students to understand the implications of work in AI to society, and to make responsible choices for when and how to apply AI techniques.

Math Requirements

Required:

- Discrete Math:
    - sets, relations, functions
    - predicate and first-order logic, logic-based proofs
- Linear Algebra:
    - Matrix operations, matrix algebra
    - Basis sets
- Probability and Statistics:

  - Basic probability theory, conditional probability, independence
  - Bayes theorem and applications of Bayes theorem
  - Expected value, basic descriptive statistics, distributions
  - Basic summary statistics and significance testing
  - All should be applied to real decision making examples with real data, not “textbook” examples

Desirable:

- Calculus-based probability and statistics
- Other topics in probability and statistics
  - Hypothesis testing, data resampling, experimental design techniques
- Optimization

Course Packaging Suggestions

Artificial Intelligence to include the following:

- AI-Introduction (4 hours)
- AI-Search: 9 hrs
- AI-KRR: 4 hrs
- AI-ML: 12 hrs
- AI-Prob: 5 hrs
- AI-SEP: 4 hrs (should be integrated throughout the course)

Prerequisites:

- CS2
- Discrete math
- Probability

Skill statement: A student who completes this course should understand the basic areas of AI and be able to understand, develop, and apply techniques in each. They should be able to solve problems using search techniques, basic Bayesian reasoning, and simple machine learning methods. They should understand the various applications of AI and associated ethical and societal implications.

Machine Learning to include the following:

- AI-ML: 32 hrs
- AI-KRR: 4 hrs
- AI-NLP: 4 hrs (selected topics, e.g., TF-IDF, bag of words, and text classification)
- AI-SEP: 4 hrs (should be integrated throughout the course)

Prerequisites:

- CS2
- Discrete math
- Probability
- Statistics
- Linear algebra (optional)

Skill statement: A student who completes this course should be able to understand, develop, and apply mechanisms for supervised learning and reinforcement learning. They should be able to select the proper machine learning algorithm for a problem, preprocess the data appropriately, apply proper evaluation techniques, and explain how to interpret the resulting models, including the model's shortcomings. They should be able to identify and compensate for biased data sets and other sources of error, and be able to explain ethical and societal implications of their application of machine learning to practical problems.

Robotics to include the following:

- AI-Robo: Robotics: 25 hrs
- SPD-D: Robot Platforms: 4 hrs (focusing on hardware, constraints/considerations, and software architectures; some other topics in SPD/Robot Platforms overlap with AI/Robotics)
- AI-Search: Search: 4 hrs (selected topics well-integrated with robotics, e.g., A* and path search)
- AI-ML: 6 hrs (selected topics well-integrated with robotics, e.g., neural networks for object recognition)
- AI-SEP: 3 hrs (should be integrated throughout the course; robotics is already a huge application, so this really should focus on societal impact and specific robotic applications)

Prerequisites:

- CS2
- Linear algebra

Skill statement: A student who completes this course should be able to understand and use robotic techniques to perceive the world using sensors, localize the robot based on features and a map, and plan paths and navigate in the world in simple robot applications. They should understand and be able to apply simple computer vision, motion planning, and forward and inverse kinematics techniques.

Data science to include the following:

- AI-ML
- Data management
- Visualization

Committee

Chair: Eric Eaton, University of Pennsylvania, Philadelphia, PA, USA

Members:

- Zachary Dodds, Harvey Mudd College, Claremont, CA, USA
- Susan L. Epstein, Hunter College and The Graduate Center of The City University of New York, New York, NY, USA
- Laura Hiatt, US Naval Research Laboratory, Washington, DC, USA
- Amruth N. Kumar, Ramapo College of New Jersey, Mahwah, USA
- Peter Norvig, Google, Mountain View, CA, USA
- Meinolf Sellmann, GE Research, Niskayuna, NY, USA
- Reid Simmons, Carnegie Mellon University, Pittsburgh, PA, USA

Contributors:

- Claudia Schulz, Thomson Reuters, Zurich, Switzerland

Algorithmic Foundations (AL)

Preamble

Algorithms and data structures are fundamental to computer science and software engineering since every theoretical computation and real-world program consists of algorithms that operate on data elements possessing an underlying structure. Selecting appropriate computational solutions to real-world problems benefits from understanding the theoretical and practical capabilities, and limitations, of available algorithms and paradigms, including their impact on the environment and society. Moreover, this understanding provides insight into the intrinsic nature of computation, computational problems, and computational problem-solving as well as possible solution techniques independent of programming language, programming paradigm, computer hardware, or other implementation aspects.

This knowledge area focuses on the nature of algorithmic computation including the concepts and skills required to design and analyze algorithms for solving real-world computational problems. It complements the implementation algorithms and data structures found in the Software Development Foundations (SDF) knowledge area. As algorithms and data structures are essential in all advanced areas of computer science, this area provides the algorithmic foundations that every computer science graduate is expected to know. Exposure to the breadth of these foundational AL topics is designed to provide students with the basis for studying additional topics in algorithmic computation in more depth and for learning advanced algorithms across a variety of knowledge areas and disciplines.

Changes since CS 2013

This area has been renamed to better reflect its foundational scope since topics in this area focus on the theoretical foundations of complexity and computability. They also provide the foundational prerequisites for advanced study in computer science. Additionally, topics focused on complexity and computability have been clearly separated into their respective knowledge units.To reinforce the important impact of computation on society, an Algorithms and Society unit has been added with the expectation that Societal, Ethical, and Professional (SEP) implications be addressed in some way during every lecture hour in the AL knowledge area.

The increase of four CS core hours acknowledges the importance of this foundational area in the computer science curriculum and returns it to the 2001 level. Despite this increase, there is a significant overlap in hours with the Software Development Fundamentals (SDF) and Mathematical Foundations (MSF) areas. There is also a complementary nature of the units in this area since, for example, linear search of an array covers topics in AL-Fundamentals and can be used to simultaneously explain AL-Complexity, e.g., O(n), and AL-Strategies, e.g. Brute-Force.
The KA hours primarily reflect topics studied in a stand-alone computational theory course and the availability of additional hours when such a course is included in the curriculum.

Core Hours

Knowledge Unit	**CS Core**	**KA Core**
Foundational Data Structures and Algorithms	14	6
Algorithmic Strategies	6	
Complexity Analysis	6	3
Computational Models and Formal Languages	6	23
Algorithms and Society	Included in SEP hours	
Total	**32**	**32**

Knowledge Units

AL-Foundational: Foundational Data Structures and Algorithms

CS Core:

1. Abstract Data Types (See also: SDF-ADT, FPL-Types: 1)
    a. Dictionary Operations (insert, delete, find)
    b. Objects (See also: FPL-OO: 2a)
2. Arrays (See also: SDF-Fundamentals, SDF-ADT)
    a. Numeric vs. Non-numeric, Character Strings
    b. Single (Vector) vs. Multidimensional (Matrix)
3. Records/Structs/Tuples (See also: FPL-Types: 1)
4. Linked Lists (See also: SDF-ADT)
    a. Single vs. double and Linear vs. Circular
5. Stacks (See also: SDF-ADT, AL-Models)
6. Queues and Dequeues (See also: SDF-ADT)
7. Hash Tables / Maps (See also: SDF-ADT)
    a. Collision Resolution and Complexity (e.g., probing, chaining, rehash)
8. Graphs (e.g., [un]directed, [a]cyclic, [un]connected, and [un]weighted) (See also: MSF-Discrete: 7)
    a. Adjacency list vs. matrix representations
9. Trees (See also: MSF-Discrete: 7)
    a. Binary, n-ary, and search trees
    b. Balanced (e.g., AVL, Red-Black)
10. Sets (See also: MSF-Discrete 1)
11. Search Algorithms (See also: SDF-Algorithms)
    a. O(n) (e.g., linear/sequential search)
    b. O($\log_2 n$) (e.g., binary search)

    c. O($\log_b n$) (e.g. depth/breadth-first tree)

12. Sorting Algorithms (e.g., stable, unstable) (See also: SDF-Algorithms)
    a. $O(n^2)$ complexity (e.g., insertion, selection),
    b. $O(n \log n)$ complexity (e.g., quicksort, merge, timsort)
13. Graph Algorithms
    a. Shortest Path (e.g., Dijkstra's, Floyd's)
    b. Minimal spanning tree (e.g., Prim's, Kruskal's)

KA Core:

1. Heaps and Priority Queues
2. Sorting Algorithms
    a. $O(n \log n)$ heapsort
    b. Pseudo O(n) complexity (e.g., bucket, counting, radix)
3. Graph Algorithms
    a. Transitive closure (e.g., Warshall's Algorithm)
    b. Topological sort
4. Matching
    a. Efficient String Matching (e.g., Boyer-Moore, Knuth-Morris-Pratt)
    b. Longest common subsequence matching
    c. Regular expression matching

Non Core:

5. Cryptography Algorithms (e.g., SHA-256) (See also: SE-Cryptography, MSF-Discrete: 5)
6. Parallel Algorithms (See also: PDC-Algorithms, FPL-Parallel)
7. Consensus algorithms (e.g., Blockchain) (See also: SE-Cryptography: 14)
    a. Proof of work vs. proof of stake (See also: SEP-Sustainability: 3)
8. Quantum computing algorithms (See also: AR-Quantum: 6)
    a. Oracle-based (e.g. Deutsch-Jozsa, Bernstein-Vazirani, Simn)
    b. Superpolynomial speed-up via QFT (e.g., Shor's algorithm)
    c. Polynomial speed-up via amplitude amplification (e.g., Grover's algorithm)

Illustrative Learning Outcomes:

CS Core:

1. For each Fundamental ADT/Data Structure in this unit:
    a. Articulate its definition, properties, representation(s), and associated ADT operations,
    b. Using a real-world example, explain step-by-step how the ADT operations associated with the data structure transform it.
2. For each of the algorithmic in this unit:
    a. Use a real-world example, show step-by-step how the algorithm operates.
3. For each of the algorithmic approach in this unit:
    a. Give a prototypical example of the approach (e.g., Quicksort for Sorting)
4. Given requirements for a real-world application, create multiple design solutions using various data structures and algorithms. Subsequently, evaluate the suitability, strengths, and weaknesses of the selected approach for satisfying the requirements.

5. Explain how collision avoidance and collision resolution is handled in hash tables.
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.

KA Core:

7. Describe the heap property and the use of heaps as an implementation of a priority queue.
8. For each of the algorithms and algorithmic approaches in the KA core topics:
    a. Give a prototypical example of the algorithm,
    b. Use a real-world example, show step-by-step how the algorithm operates.

AL-Strategies: Algorithmic Strategies

CS Core:

1. Paradigms
    a. Brute-Force (e.g., linear search, selection sort, traveling salesperson, knapsack)
    b. Decrease-and-Conquer
        i. By a Constant (e.g., insertion sort, topological sort),
        ii. By a Constant Factor (e.g., binary search),
        iii. By a Variable Size (e.g., Euclid's algorithm)
    c. Divide-and-Conquer (e.g., Binary Search, Quicksort, Mergesort, Strassen's)
    d. Greedy (e.g., Dijkstra's, Kruskal's)
    e. Transform-and-Conquer
        i. Instance simplification (e.g. find duplicates via list presort)
        ii. Representation change (e.g., heapsort)
        iii. Problem reduction (e.g., least-common-multiple, linear programming)
        iv. Dynamic Programming (e.g., Floyd's)
    f. Space vs. Time Tradeoffs (e.g., hashing) (See also: AL-Fundamentals)
2. Handling Exponential Growth (e.g., heuristics, A*, branch-and-bound, backtracking)
3. Iteration vs. Recursion (e.g., factorial) (See also: MSF-Discrete: 2)

KA Core:

4. Paradiams
    a. Approximation Algorithms
    b. Iterative improvement (e.g., Ford-Fulkerson, Simplex)
    c. Randomized/Stochastic Algorithms (e.g., Max-Cut, Balls and Bins)

Non Core:

5. Quantum computing (See also AL-Fundamentals: 8, AL-Models: 8)

Illustrative Learning Outcomes
CS Core:

1. For each of the paradigms in this unit

    a. Articulate its definitional characteristics,
    b. Give an example that demonstrates the paradigm including explaining how this example satisfies the paradigm’s characteristics
2. For each of the algorithms in the AL-Fundamentals unit:
    a. Describe the paradigm used by the algorithm and how it exemplifies this paradigm
3. Given an algorithm, describe the paradigm used by the algorithm and how it exemplifies this paradigm
4. Give a real-world problem, determine appropriate algorithmic paradigms and algorithms from these paradigms that address the problem including considering the tradeoffs among the paradigms and algorithms selected.
5. Give an example of an iterative and a recursive algorithm that solves the same problem including explaining the benefits and disadvantages of each approach.
6. Determine if a greedy approach leads to an optimal solution.
7. Explain at least one approach for addressing a computational problem whose algorithmic solution is exponential.

AL-Complexity: Complexity

CS Core:

1. Complexity Analysis Framework
    a. Best, average, and worst case performance of an algorithm
    b. Empirical and Relative (Order of Growth) Measurements
    c. Input Size and Primitive Operations
    d. Time and Space Efficiency
2. Asymptotic complexity analysis (average and worst case bounds)
    a. Big-O, Big-Omega, and Big-Theta formal notations
    b. Foundational complexity classes and representative examples/problems
        i. $O(1)$ *Constant* (e.g., Array Access)
        ii. $O(\log_2 n)$ *Logarithmic* (e.g., Binary Search)
        iii. $O(n)$ *Linear* (e.g., Linear Search)
        iv. $O(n \log_2 n)$ *Log Linear* (e.g., Mergesort
        v. $O(n^2)$ *Quadratic* (e.g., Selection Sort)
        vi. $O(n^3)$ *Cubic* (e.g., Gaussian Elimination)
        vii. $O(2^n)$ *Exponential* (e.g., Knapsack, SAT, TSP, All Subsets)
        viii. $O(n!)$ *Factorial* (e.g., Hamiltonian Circuit, All Permutations)
3. Empirical measurements of performance
4. Tractability and Intractability
    a. P, NP and NP-Complete complexity classes
    b. NP-Complete problems (e.g., SAT, Knapsack, TSP)
    c. Reductions
5. Time and space trade-offs in algorithms.

KA Core:

6. Little-o and Little-Omega notations
7. Recursive Analysis: (e.g., recurrence relations, Master theorem, substitution)
8. Amortized Analysis
9. Turing Machine-Based Models of Complexity
    a. Time complexity (See also: AL-Models)
        i. P, NP, NP-C, and EXP classes
        ii. Cook-Levin Theorem
    b. Space Complexity
        i. NSpace and PSpace
        ii. Savitch's Theorem

Illustrative Learning Outcomes
CS Core:

1. Explain what is meant by "best", "average", and "worst" case behavior of an algorithm..
2. State and explain the formal definitions of Big-O, Big-Omega, and Big-Theta notations and how they are used to describe the amount of work done by an algorithm.
3. Compare and contrast each of the foundational complexity classes listed in this unit.
4. For each foundational complexity class in this unit:
    a. Give an algorithm that demonstrates the associated runtime complexity.
5. For each algorithm in the AL-Fundamentals unit:
    a. Give its runtime complexity class and explain why it belongs to this class.
6. Determine informally the foundational complexity class of simple algorithms.
7. Perform empirical studies to determine and validate hypotheses about the runtime complexity of various algorithms by running algorithms on input of various sizes and comparing actual performance to the theoretical analysis.
8. Give examples that illustrate time-space trade-offs of algorithms.
9. Explain how tree balance affects the efficiency of various binary search tree operations.
10. Explain to a non-technical audience the significance of tractable versus intractable algorithms using an intuitive explanation of Big-O complexity.
11. Explain the significance of NP-Completeness.
12. Describe NP-Hard as a lower bound and NP as an upper bound for NP-Completeness.
13. Provide examples of NP-complete problems.

KA Core:

14. Use recurrence relations to determine the time complexity of recursively defined algorithms.
15. Solve elementary recurrence relations using some form of the Master Theorem.
16. Use Big-O notation to give upper case bounds on time/space complexity of algorithms.
17. Explain the Cook-Levin Theorem and the NP-Completeness of SAT.
18. Define the classes P and NP.
19. Prove that a problem is NP-Complete by reducing a classic known NP-C problem to it (e.g., 3SAT and Clique)
20. Define the P-space class and its relation to the EXP class.

AL-Models: Computational Models and Formal Languages

CS Core:

1. Formal Automata
    a. Finite State
    b. Pushdown (See also: AL-Fundamentals: 5, SDF-ADT)
    c. Linear Bounded
    d. Turing Machine
2. Formal Languages, Grammars and Chomsky Hierarchy (See also: FPL-H Translation, FPL-J Syntax)
    a. Regular (Type-3)
        i. Regular Expressions
    b. Context-Free (Type-2)
    c. Context-Sensitive (Type-1)
    d. Recursively Enumerable (Type-0)
3. Relations among formal automata, languages, and grammars
4. Decidability, (un)computability, and halting
5. The Church-Turing Thesis
6. Algorithmic Correctness
    a. Invariants (e.g., in: iteration, recursion, tree search)

KA Core:

1. Deterministic and nondeterministic automata
2. Pumping Lemma Proofs (See also: MSF-Discrete: 3)
    a. Finite State/Regular
    b. Pushdown Automata/Context-Free
3. Decidability
    a. Arithmetization and Diagonalization (See also: MSF-Discrete: 1)
4. Reducibility and reductions
5. Time Complexity based on Turing Machine
6. Space Complexity (e.g., PSPACE, Savitch's Theorem)
7. Equivalent Models of Algorithmic Computation
    a. Turing Machines and Variations (e.g., multi-tape, non-deterministic)
    b. Lambda Calculus (See also: FPL-Functional)
    c. Mu-Recursive Functions

Non Core:

8. Quantum Computation (See also: AR-Quantum)
    a. Postulates of quantum mechanics
        i. State Space
        ii. State Evolution
        iii. State Composition
        iv. State Measurement
    b. Column vector representations of Qubits

c. Matrix representations of quantum operations
d. Quantum Gates (e.g., XNOT, CNOT)

Illustrative Learning Outcomes
CS Core:

1. For each formal automata in this unit:
    a. Articulate its definition comparing its characteristics with this unit's other automata,
    b. Using an example, explain step-by-step how the automata operates on input including whether it accepts the associated input,
    c. Give an example of inputs that can and cannot be accepted by the automata.
2. Given a real-world problem, design an appropriate automaton that addresses the problem.
3. Design a Regular Expression to accept a sentence from a Regular language.
4. Explain the difference between Regular Expressions (Type-3 acceptors) and Re-Ex pattern matchers (Type-2 acceptors) used in programming languages.
5. For each formal language/grammar in this unit
    a. Articulate its definition comparing its characteristics with the others in this unit,
    b. Give an example of inputs that can and cannot be accepted by the language/grammar.
6. Describe a univTuring Machine.
7. Explain how decidability and computability for various automata are related.
8. Explain the Halting problem, why it has no algorithmic solution, and its significance for real-world algorithmic computation.
9. Give examples of classic uncomputable problems.
10. Explain the Church-Turing Thesis and its significance for algorithmic computation.
11. Explain how invariants assist in proving the correctness of an algorithm as a formal model.

Illustrative Learning Outcomes
KA Core:

1. For each formal automata in this unit
    a. Compare/contrast its deterministic and nondeterministic capabilities.
2. Use a pumping lemma to prove the limitations of Finite State and Pushdown automata.
3. Use arithmetization and diagonalization to prove Turing Machine Halting/Undecidability.
4. Explain a reduction such as between Halting and Undecidability of the language accepted by a Turing Machine, where one has been previously proven by diagonalization.
5. Convert among equivalently powerful notations for a language, including among DFAs, NFAs, and regular expressions, and between PDAs and CFGs
6. Explain Rice's Theorem and its significance.
7. Give an example proof of a problem that is uncomputable by reducing a classic known uncomputable problem to it
8. Explain the Primitive and General Recursive functions (zero, successor, selection, primitive recursion, composition, and Mu), their significance, and Turing Machine implementations.
9. Explain how computation is performed in Lambda Calculus (e.g., Alpha Conversion and Beta-Reduction)

Non Core:

10. For a quantum system give examples that explain the following postulates:
    a. State Space: system state represented as a unit vector in Hilbert space,
    b. State Evolution: the use of unitary operators to evolve system state,
    c. State Composition: the use of tensor product to compose systems states,
    d. State Measurement: the probabilistic output of measuring a system state.
11. Explain the operation of a quantum XNOT or CNOT gate on a quantum bit represented as a matrix and column vector respectively

AL-SEP: Society, Ethics, and Professionalism

CS Core: (See also: SEP-Context, SEP-Sustainability)

1. Social, Ethical, and Secure Algorithms
2. Algorithmic Fairness (e.g., Differential Privacy)
3. Accountability/Transparency
4. Responsible algorithms
5. Economic and other impacts of inefficient algorithms
6. Sustainability

Illustrative Learning Outcomes
CS Core:

1. Devise algorithmic solutions to real-world societal problems, such as routing an ambulance to a hospital
2. Predict and explain the impact that an algorithm may have on the environment and society when used to solve real-world problems taking into account that it can affect different societal groups in different ways and the algorithm’s sustainability.
3. Prepare a presentation that justifies the selection of appropriate data structures and/or algorithms to solve a given real-world industry problem.
4. Give an example that articulates how differential privacy protects knowledge of an individual’s data.
5. Describe the environmental impacts of design choices that relate to algorithm design.
6. Discuss the tradeoffs involved in proof-of-work and proof-of-stake algorithms.

Professional Dispositions

- **Meticulous:** As an algorithm is a formal solution to a computational problem, attention to detail is important when developing and combining algorithms.
- **Persistent:** As developing algorithmic solutions to computational problems can be challenging, computer scientists must be resolute in pursuing such solutions
- **Inventive:** As computer scientists develop algorithmic solutions to real-world problems, they must be inventive in developing solutions to these problems.

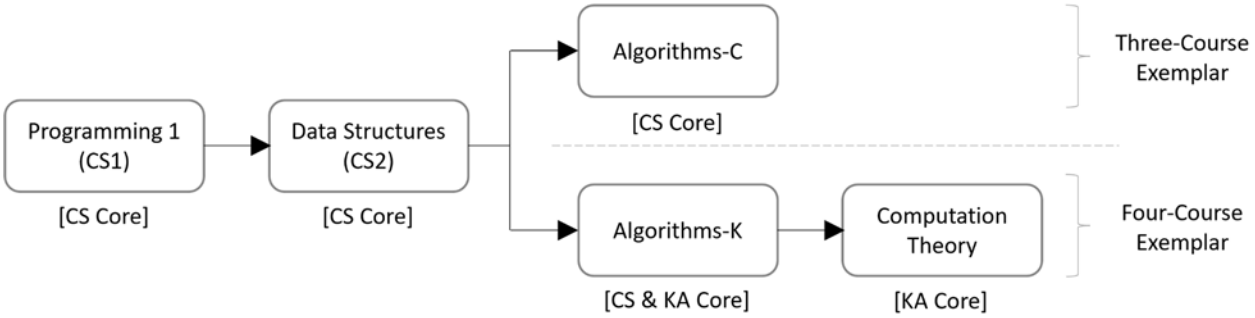
Math Requirements

Required:

- MSF-Discrete

Course Packaging Suggestions

As depicted in the following figure, the committee envisions two common approaches for addressing foundational AL topics in CS courses. Both approaches included required introductory Programming (CS1) and Data Structures (CS2) courses. In the three-course approach, all CS Core topics are covered. Alternatively, in the four-course approach, AL-Model unit CS and KA core topics are addressed in a Computational Theory focused course, which leaves room to address additional KA topics in the third Algorithms course. Both approaches assume Big-O analysis is introduced in the Data Structures (CS2) course and that graphs are taught in the third Algorithms course. The committee recognizes that there are many different approaches for packaging AL topics into courses including, for example, introducing graphs in CS2 Data Structures, Backtracking in an AI course, and AL-Model topics in a theory course which also addresses FPL topics. The given example is simply one way to cover the entire AL CS Core in three introductory courses.



Courses Common to Three and Four Course Exemplars

Programming 1 (CS1)

- AL-Foundational: Fundamental Data Structures and Algorithms (2 hours)
    - Arrays and Strings
    - Linear Search

Note: the following AL topics are demonstrated in CS1, but not explicitly taught as such:

- AL-Strategies: Algorithmic Strategies
    - Brute Force (e.g., linear search)
    - Iteration (e.g., linear search)
- AL-Complexity: Complexity Analysis
    - O(1) and O(n) runtime complexities

Data Structures (CS2)

- AL-Foundational: Fundamental Data Structures and Algorithms (12 hours)
    - Abstract Data Types and Operations (ADTs)
    - Binary Search

  - Multi-dimensional Arrays
  - Linked Lists
  - Hash Tables/Maps including conflict resolution strategies
  - Stacks, Queues, and Dequeues
  - Trees: Binary, Ordered, Breadth- and Depth-first search
  - An $O(n^2)$ sort, (e.g., Selection Sort)
  - An $O(n \log n)$ sort (e.g., Quicksort, Mergesort)

- AL-Strategies: Algorithmic Strategies (3 hour)
  - Brute Force (e.g., selection sort)
  - Decrease-and-Conquer (e.g., depth/breadth tree search)
  - Divide-and-Conquer (e.g., mergesort, quicksort)
  - Recursive (e.g., depth/breadth-first tree/graph search, factorial)
  - Space vs. Time tradeoff (e.g., hashing)
- AL-Complexity: Complexity Analysis
  - Asymptotic complexity analysis
  - Empirical measurements of performance
  - Time-and-space tradeoffs (e.g., hashing)
- AL-SEP: Algorithms and Society Algorithms and Society (2 hours)

Three Course Exemplar Approach
Algorithms-C

- AL-Foundational: Fundamental Data Structures and Algorithms (3 hours)
- AL-Strategies: Algorithmic Strategies (1 hour)
  - Brute Force (e.g., traveling salesperson, knapsack)
  - Decrease-andConquer (e.g., topological sort
  - Divide-andConquer (e.g., Strassen's algorithm)
  - Dynamic Programming (e.g. Warshall's, Floyd's, Bellman-Ford)
  - Greedy (e.g., Dijkstra's, Kruskal's)
  - Heuristic (e.g., A*)
  - Transform-and-Conquer/Reduction (e.g., heapsort, trees (2-3, AVL, Red-Black))
- AL-Models: Computational Models (6 hours)
  - All CS core topics

Four Course Exemplar Approach
Algorithms-C

- All topics from Algorithms-C course plus AL-Foundational KA Core (6 hours)

Computation Theory

- AL-Complexity: Complexity Analysis (3 hours)
  - Turing Machine-based models of complexity (P, NP, and NP-C classes)
  - Space complexity (NSpace, PSpace Savitch' Theorem)
- AL-Models: Computational Models (29 hours)
  - All CS and KA Core topics

Committee

Chair: Richard Blumenthal, Regis University, Denver, Colorado, USA

Members:

- Cathy Bareiss, Bethel University, Mishawaka, Minnesota, USA
- Tom Blanchet, Hillman Companies Inc., Boulder, Colorado, USA
- Doug Lea, State University of New York at Oswego, Oswego, New York, USA
- Sara Miner More, John Hopkins University, Baltimore, Maryland, USA
- Mia Minnes, University of California San Diego, California, USA
- Atri Rudra, University at Buffalo, Buffalo, New York, USA
- Christian Servin, El Paso Community College, El Paso, Texas, USA

Architecture and Organization (AR)

Preamble

Computing professionals spend considerable time writing efficient code to solve a particular problem in an application domain. Parallelism and heterogeneity at the hardware system level have been increasingly utilized to meet performance requirements in almost all systems, including most commodity hardware. This departure from sequential processing demands a more in-depth understanding of the underlying computer architectures. Architecture can no longer be treated as a *black box* where principles from one can be applied to another. Instead, programmers should look inside the black box and use specific components to improve system performance and energy efficiency.

The Architecture and Organization (AR) Knowledge Area aims to develop a deeper understanding of the hardware environments upon which almost all computing is based and the relevant interfaces provided to higher software layers. The target hardware comprises low-end embedded system processors up to high-end enterprise multiprocessors.

The topics in this knowledge area will benefit students by enabling them to appreciate the fundamental architectural principles of modern computer systems, including the challenge of harnessing parallelism to sustain performance and energy improvements into the future. This KA will help computer science students depart from the black box approach and become more aware of the underlying computer system and the efficiencies that specific architectures can achieve.

Changes since CS 2013

This KA has changed slightly since the CS2013 report. Changes and additions are summarized as follows:

- Topics have been revised, particularly in the Knowledge Units AR/Memory Hierarchy and AR/Performance and Energy Efficiency. This change supports recent advances in memory caching and energy consumption.
- To address emerging topics in Computer Architecture, the newly created KU AR/Heterogeneous Architectures covers the introductory level: In-Memory processing (PIM), domain-specific architectures (e.g. neural network processors) - and related topics.
- Quantum computing, particularly the development of quantum processor architectures, has been gathering pace recently. The new KU AR/Quantum Architectures offer a "toolbox" covering introductory topics in this important emerging area.
- Knowledge units have been merged to better deal with overlaps:
    - AR/Multiprocessing and Alternative Architectures were merged into newly created AR/Heterogeneous Architectures.

Overview

Core Hours

Knowledge Unit	**CS Core**	**KA Core**
Digital Logic and Digital Systems		3
Machine-Level Data Representation	1	
Assembly Level Machine Organization	1	2
Memory Hierarchy	6	
Interfacing and Communication	1	
Functional Organization		2
Performance and Energy Efficiency		3
Heterogeneous Architectures		3
Quantum Architectures		3
Total	**9**	**16**

Knowledge Units

AR-A / AR-Logic: Digital Logic and Digital Systems

KA Core:

1. Overview and history of computer architecture
2. Combinational vs sequential logic/field programmable gate arrays (FPGAs)
    a. Fundamental combinational
    b. Sequential logic building block
3. Functional hardware and software multi-layer architecture
4. Computer-aided design tools that process hardware and architectural representations
5. High-level synthesis
    a. Register transfer notation
    b. Hardware description language (e.g. Verilog/VHDL/Chisel)
6. System-on-chip (SoC) design flow
7. Physical constraints
    a. Gate delays
    b. Fan-in and fan-out
    c. Energy/power
    d. Speed of light

Illustrative Learning Outcomes

KA Core:

1. Comment on the progression of computer technology components from vacuum tubes to VLSI, from mainframe computer architectures to the organization of warehouse-scale computers.
2. Comment on parallelism and data dependencies between and within components in a modern heterogeneous computer architecture.
3. Explain how the "power wall" makes it challenging to harness parallelism.
4. Propose the design of basic building blocks for a computer: arithmetic-logic unit (gate-level), registers (gate-level), central processing unit (register transfer-level), and memory (register transfer-level).
5. Evaluate simple building blocks (e.g., arithmetic-logic unit, registers, movement between registers) of a simple computer design.
6. Validate the timing diagram behavior of a pipelined processor, identifying data dependency issues.

AR-B / AR-Representation: Machine-Level Data Representation

CS Core:

1. Bits, bytes, and words
2. Numeric data representation and number bases (See also: GIT-Visualization)
    a. Fixed-point
    b. Floating-point
3. Signed and twos-complement representations
4. Representation of non-numeric data
5. Representation of records and arrays

Illustrative Learning Outcomes

CS Core:

1. Comment on the reasons why everything is data in computers, including instructions.
2. Follow the diagram and annotate the regions where fixed-length number representations affect accuracy and precision.
3. Comment on how negative integers are stored in sign-magnitude and twos-complement representations.
4. Articulate with plausible justification how different formats can represent numerical data.
5. Explain the bit-level representation of non-numeric data, such as characters, strings, records, and arrays.
6. Translate numerical data from one format to another.
7. Explain how a single adder (without overflow detection) can handle both signed (two's complement) and unsigned (binary) input without "knowing" which format a given input is using.

AR-C / AR-Assembly: Assembly Level Machine Organization

CS Core:

1. von Neumann machine architecture
2. Control unit; instruction fetch, decode, and execution

3. Introduction to SIMD vs. MIMD and the Flynn taxonomy (See also: PDC-A: Programs and Execution)
4. Shared memory multiprocessors/multicore organization

KA Core:

5. Instruction set architecture (ISA) (e.g., x86, ARM and RISC-V)
    a. Fixed vs. variable-width instruction sets
    b. Instruction formats
    c. Data manipulation, control, I/O
    d. Addressing modes
    e. Machine language programming
    f. Assembly language programming
6. Subroutine call and return mechanisms (xref PL/language translation and execution)
7. I/O and interrupts
8. Heap, static, stack and code segments

Illustrative Learning Outcomes

CS Core:

1. Contextualize the classical von Neumann functional units in embedded systems, particularly on-chip and off-chip memory.
2. Comment on how instruction is executed in a classical von Neumann machine, with extensions for threads, multiprocessor synchronization, and SIMD execution.
3. Annotate an example diagram with instruction-level parallelism and hazards to comment on how they are managed in typical processor pipelines.

KA Core:

4. Comment on how instructions are represented at the machine level and in the context of a symbolic assembler.
5. Map an example of high-level language patterns into assembly/machine language notations.
6. Comment on different instruction formats, such as addresses per instruction and variable-length vs fixed-length formats.
7. Follow a subroutine diagram to comment on how subroutine calls are handled at the assembly level.
8. Comment on basic concepts of interrupts and I/O operations.
9. Code a simple assembly language program for string array processing and manipulation.

AR-D / AR-Memory: Memory Hierarchy

CS Core:

1. Memory hierarchy: the importance of temporal and spatial locality (See also OS-F: Memory Management)
2. Main memory organization and operations
3. Persistent memory (e.g. SSD, standard disks)

4. Latency, cycle time, bandwidth and interleaving
5. Cache memories
    a. Address mapping
    b. Block size
    c. Replacement and store policy
6. Multiprocessor cache coherence
7. Virtual memory (hardware support, cross-reference OS/Virtual Memory) (See also: OS-F: Memory Management)
8. Fault handling and reliability
9. Reliability (cross-reference SF/Reliability through Redundancy) (See also: SF-F: System reliability)
    a. Error coding
    b. Data compression (See also: AL-Fundamentals.3)
    c. Data integrity

KA Core:

10. Non-von Neumann Architectures
    a. Processing In-Memory (PIM)

Illustrative Learning Outcomes

CS Core:

1. Using a memory system diagram, detect the main types of memory technology (e.g., SRAM, DRAM) and their relative cost and performance.
2. Measure the effect of memory latency on running time.
3. Enumerate the functions of a system with virtual memory management.
4. Compute average memory access time under various cache and memory configurations and mixes of instruction and data references.

AR-E / AR-IO: Interfacing and Communication

CS Core:

1. I/O fundamentals
    a. Handshaking and buffering
    b. Programmed I/O
    c. Interrupt-driven I/O
2. Interrupt structures: vectored and prioritized, interrupt acknowledgement
3. I/O devices (e.g., mouse, keyboard, display, camera, sensors, actuators) (see also: GIT-Fundamental Concepts, GIT-Interaction)
4. External storage, physical organization and drives
5. Buses fundamentals
    a. Bus protocols
    b. Arbitration
    c. Direct-memory access (DMA)

Illustrative Learning Outcomes

CS Core:

1. Follow an interrupt control diagram to comment on how interrupts are used to implement I/O control and data transfers.
2. Enumerate various types of buses in a computer system.
3. List the advantages of magnetic disks and contrast them with the advantages of solid-state disks.

AR-F / AR-Organization: Functional Organization

KA Core:

1. Implementation of simple datapaths, including instruction pipelining, hazard detection and resolution (e.g., stalls, forwarding)
2. Control unit
    a. Hardwired implementation
    b. Microprogrammed realization
3. Instruction pipelining
4. Introduction to instruction-level parallelism (ILP)

Illustrative Learning Outcomes
KA Core:

1. Validate alternative implementation of datapaths in modern computer architectures.
2. Propose a set of control signals for adding two integers using hardwired and microprogrammed implementations.
3. Comment on instruction-level parallelism using pipelining and significant hazards that may occur.
4. Design a complete processor, including datapath and control.
5. Compute the average cycles per instruction for a given processor and memory system implementation.

AR-G / AR-Performance-Energy: Performance and Energy Efficiency

KA Core:

1. Performance-energy evaluation (introduction): performance, power consumption, memory and communication costs
2. Branch prediction, speculative execution, out-of-order execution, Tomasulo's algorithm
3. Prefetching
4. Enhancements for vector processors and GPUs
5. Hardware support for Multithreading
    a. Race conditions
    b. Lock implementations
    c. Point-to-point synchronization
    d. Barrier implementation
6. Scalability

7. Alternative architectures, such as VLIW/EPIC, accelerators and other special-purpose processors
8. Dynamic voltage and frequency scaling (DVFS)
9. Dark Silicon

Illustrative Learning Outcomes
KA Core:

1. Comment on evaluation metrics for performance and energy efficiency.
2. Follow a speculative execution diagram and write about the decisions that can be made.
3. Build a GPU performance-watt benchmarking diagram.
4. Code a multi-threaded program that adds (in parallel) elements of two integer vectors.
5. Propose a set of design choices for alternative architectures.
6. Comment on key concepts associated with dynamic voltage and frequency scaling.
7. Measure energy savings improvement for an 8-bit integer quantization compared to a 32-bit quantization.

AR-H / AR-Heterogeneity: Heterogeneous Architectures

KA Core:

1. SIMD and MIMD architectures (e.g. General-Purpose GPUs, TPUs and NPUs) (See also: SPD-C: Mobile Platforms, GIT-Shading)
2. Heterogeneous memory system
   a. Shared memory versus distributed memory
   b. Volatile vs non-volatile memory
   c. Coherence protocols
3. Domain-Specific Architectures (DSAs) (AI-D: Machine Learning; HCI-B: Accountability and Responsibility in Design)
   a. Machine Learning Accelerator
   b. In-networking computing
   c. Embedded systems for emerging applications
   d. Neuromorphic computing
4. Packaging and integration solutions such as 3DIC and Chiplets

Illustrative Learning Outcomes
KA Core

1. Follow a system diagram with alternative parallel architectures, e.g. SIMD and MIMD, and annotate the key differences.
2. Tell what memory-management issues are found in multiprocessors that are not present in uniprocessors and how these issues might be resolved.
3. Validate the differences between memory backplane, processor memory interconnect, and remote memory via networks, their implications for access latency and their impact on program performance.

4. Tell how you would determine when to use a domain-specific accelerator instead of a general-purpose CPU.
5. Enumerate key differences in architectural design principles between a vector and scalar-based processing unit.
6. List the advantages and disadvantages of PIM architectures.

AR-I / AR-Quantum: Quantum Architectures

KA Core:

1. Principles
    a. The wave-particle duality principle
    b. The uncertainty principle in the double-slit experiment
    c. What is a Qubit? Superposition and measurement. Photons as qubits.
    d. Systems of two qubits. Entanglement. Bell states. The No-Signaling theorem.
2. Axioms of QM: superposition principle, measurement axiom, unitary evolution
3. Single qubit gates for the circuit model of quantum computation: X, Z, H.
4. Two qubit gates and tensor products. Working with matrices.
5. The No-Cloning Theorem. The Quantum Teleportation protocol.
6. Algorithms
    a. Simple quantum algorithms: Bernstein-Vazirani, Simon's algorithm.
    b. Implementing Deutsch-Josza with Mach-Zehnder Interferometers.
    c. Quantum factoring (Shor's Algorithm)
    d. Quantum search (Grover's Algorithm)
7. Implementation aspects
    a. The physical implementation of qubits
    b. Classical control of a Quantum Processing Unit (QPU)
    c. Error mitigation and control. NISQ and beyond.
    d. Measurement approaches
8. Emerging Applications
    a. Post-quantum encryption
    b. The Quantum Internet
    c. Adiabatic quantum computation (AQC) and quantum annealing

Illustrative Learning Outcomes

KA Core:

1. Comment on a quantum object produced as a particle, propagates like a wave and is detected as a particle with a probability distribution corresponding to the wave.
2. Follow the diagram and comment on the quantum-level nature that is inherently probabilistic.
3. Articulate your view on entanglement that can be used to create non-classical correlations, but there is no way to use quantum entanglement to send messages faster than the speed of light.
4. Comment on quantum parallelism and the role of constructive vs destructive interference in quantum algorithms given the probabilistic nature of measurement(s).
5. Annotate a code snippet providing the role of quantum Fourier transform (QFT) in Shor's algorithm.

6. Code Shor's algorithm in a simulator and document your code, highlighting the classical components and aspects of Shor's algorithm.
7. Enumerate the specifics of each qubit modality (e.g., trapped ion, superconducting, silicon spin, photonic, quantum dot, neutral atom, topological, color center, electron-on-helium, etc.),
8. Contextualize the differences between AQC and the gate model of quantum computation and which kind of problems each is better suited to solve.
9. Comment on the statement: a QPU is a heterogeneous multicore architecture like an FPGA or a GPU.

Dispositions

- Self-directed: students should increasingly become self-motivated to acquire complementary knowledge.
- Proactive: students should exercise control and antecipate issues related to the underlying computer system.

Math Requirements

Required:

- Discrete Math: Sets, Relations, Logical Operations, Number Theory, Boolean Algebra
- Linear Algebra: Arithmetic Operations, Matrix operations
- Logarithms, Limits

Desired:

- Math/Physics for Quantum Computing: basic probability, trigonometry, simple vector spaces, complex numbers, Euler's formula
- System performance evaluation: probability and factorial experiment design.

Course Packaging Suggestions

Computer Architecture - Introductory Course to include the following:

- SEP-History [2 hours]
- AR-B / AR-Representation: Machine-Level Data Representation [2 hours]
- AR-C / AR-Assembly: Assembly Level Machine Organization [2 hours]
- AR-D / AR-Memory: Memory Hierarchy [10 hours]
- OS/Memory Management [10 hours]
- AR-E / AR-IO: Interfacing and Communication [4 hours]
- AR-H / AT-Heterogeneity: Heterogeneous Architectures [5 hours]
- PD/Programs and Executions [4 hours]
- SEP/Methods for Ethical Analysis [3 hours]

Pre-requisites:

- Discrete Math: Sets, Relations, Logical Operations, Number Theory, Basic Programming

Skill statement:

- A student who completes this course should be able to understand the fundamental architectures of modern computer systems, including the challenge of memory caches, memory management and pipelining.

Systems Course to include the following:

- SEP-History [2 hours]
- SEP/Privacy and Civil Liberties [3 hours]
- SF-H: System Design [2 hours]
- SF-F: System Reliability [2 hours]
- PD/Algorithms and applications [4 hours]
- AR-H / AR-Heterogeneity: Heterogeneous Architectures [4 hours]
- OS/Roles and Purpose of Operating Systems [3 hours]
- AR-D / AR-Memory: /Memory Hierarchy [7 hours]
- AR-G / AR-Performance-energy: Performance and Energy Efficiency [5 hours]
- NC/Networked Applications [5 hours]

Pre-requisites:

- Discrete Math: Sets, Relations, Logical Operations, Number Theory

Skill statement:

- A student who completes this course should be able to appreciate the advanced architectural aspects of modern computer systems, including the challenge of heterogeneous architectures and the required hardware and software interfaces to improve the performance and energy footprint of applications.

Computer Architecture - Advanced Topics Course to include the following:

- AR-A / AR-Logic.1 Digital Logic and Digital Systems/Overview and history of computer architecture [4 hours]
- SF-E: Performance Evaluation [4 hours]
- PD/Algorithms and applications [4 hours]
- AR-H / AR-Heterogeneity: Heterogeneous Architectures [4 hours]
- AR-G / AR-Performance-Energy: Performance and Energy Efficiency [5 hours]
- AR-C: Cryptography [4 hours]
- AR-I / AR-Quantum: Quantum Architectures [4 hours]

Pre-requisites:

- Discrete Math: Sets, Relations, Logical Operations, Number Theory, Probability, Linear Algebra

Skill statement:

- A student who completes this course should be able to appreciate how computer architectures evolved into today's heterogeneous systems and to what extent past design choices can influence the design of future high-performance computing

Committee

Chair: Marcelo Pias, Federal University of Rio Grande (FURG), Rio Grande, RS, Brazil

Members:

- Brett A. Becker, University College Dublin, Dublin, Ireland
- Mohamed Zahran, New York University, New York, NY, USA
- Monica D. Anderson, University of Alabama, Tuscaloosa, AL, USA
- Qiao Xiang, Xiamen University, China
- Adrian German, Indiana University, Bloomington, IN, USA

Data Management (DM)

Preamble

Each area of computer science can be described as "The study of algorithms and data structures to ..." In this case the blank is filled in with "deal with persistent data sets; frequently too large to fit in primary memory."
Since the mid-1970's this has meant an almost exclusive study of relational database systems.
Depending on institutional context, students have studied, in varying proportions:

- Data modeling and database design: e.g., E-R Data model, relational model, normalization theory
- Query construction: e.g., relational algebra, SQL
- Query processing: e.g., indices (B+tree, hash), algorithms (e.g., external sorting, select, project, join), query optimization (transformations, index selection)
- DBMS internals: e.g., concurrency/locking, transaction management, buffer management

Today's graduates are expected to possess DBMS user (as opposed to implementor) skills. These primarily include data modeling and query construction; ability to take an unorganized collection of data, organize it using a DBMS, and access/update the collection via queries.
Additionally, students need to study:

- The role data plays in an organization. This includes:
  - The Data Life Cycle: Creation-Processing-Review/Reporting-Retention/Retrieval-Destruction.
  - The social/legal aspects of data collections: e.g., scale, data privacy, database privacy (compliance) by design, de-identification, ownership, reliability, database security, and intended and unintended applications.
- Emerging and advanced technologies that are augmenting/replacing traditional relational systems, particularly those used to support (big) data analytics: NoSQL (e.g., JSON, XML, key-value store databases), cloud databases, MapReduce, and dataframes.

We recognize the existing and emerging roles for those involved with data management, which include:

- Product feature engineers: those who use both SQL and NoSQL operational databases.
- Analytical Engineers/Data Engineers: those who write analytical SQL, Python, and Scala code to build data assets for business groups.
- Business Analysts: those who build/manage data most frequently with Excel spreadsheets.
- Data Infrastructure Engineers: those who implement a data management system (e.g., OLTP).
- "Everyone:" those who produce or consume data need to understand the associated social, ethical, and professional issues.

One role that transcends all of the above categories is that of data custodian. Previously, data was seen as a resource to be managed (Information Systems Management) just like other enterprise resources. Today, data is seen in a larger context. Data about customers can now be seen as belonging to (or in some national contexts, as owned by) those customers. There is now an accepted understanding that the safe and ethical storage, and use, of institutional data is part of being a responsible data custodian.

Furthermore, we acknowledge the tension between a curricular focus on professional preparation versus the study of a knowledge area as a scientific endeavor. This is particularly true with Data Management. For example, proving (or at least knowing) the completeness of Armstrong's Axioms is fundamental in functional dependency theory. However, the vast majority of computer science graduates will never utilize this concept during their professional careers. The same can be said for many other topics in the Data Management canon. Conversely, if our graduates can only normalize data into Boyce-Codd normal form (using an automated tool) and write SQL queries, without understanding the role that indices play in efficient query execution, we have done a disservice.

To this end, the number of CS Core hours is relatively small relative to the KA Core hours. Hopefully, this will allow institutions with differing contexts to customize their curricula appropriately. For some, the efficient storage and access of data is primary and independent of how the data is ultimately used - institutional context with a focus on OLTP implementation. For others, what is “under the hood” is less important than the programmatic access to already designed databases - institutional context with a focus on product feature engineers/data scientists.

Regardless of how an institution manages this tension we wish to give voice to one of the ironies of computer science curricula. Students typically spend the majority of their educational career reading (and writing) data from a file or interactively, while outside of the academy the lion's share of data, by a huge margin, comes from databases accessed programmatically. Perhaps in the not too distant future students will learn programmatic database access early on and then continue this practice as they progress through their curriculum.

Finally, we understand that while Data Management is orthogonal to Cybersecurity and SEP (Society, Ethics, and Professionalism), it is also ground zero for these (and other) knowledge areas. When designing persistent data stores, the question of what should be stored must be examined from both a legal and ethical perspective. Are there privacy concerns? And finally, how well protected is the data?

Changes since CS 2013

- Rename the knowledge unit from Information Management to Data Management. This renaming does not represent any kind of philosophical shift. It is simply an effort to avoid confusion with Information Systems and Information Technology curricula.
- Inclusion of NoSQL approaches and MapReduce as CS Core topics.
- Increased attention SEP and SEC topics in both the CS Core and KA Core areas.

Core Hours

Knowledge Unit	**CS Core Hours**	**KA Core Hours**
The Role of Data	2	
Core Database Systems Concepts	2	1
Data Modeling	2	3
Relational Databases	1	3

Query Construction	2	4
Query Processing		4
DBMS Internals		4
NoSQL Systems		2
Data Security & Privacy		
Data Analytics		
Distributed Databases/Cloud Computing		
Semi-structured and Unstructured Databases		
Society, Ethics, and Professionalism		
Total	**9**	**21**

Knowledge Units

DM-Data: The Role of Data and the Data Life Cycle

CS Core:

1. The Data Life Cycle: Creation-Processing-Review/Reporting-Retention/Retrieval-Destruction (See also: SEP-Social-Context, SEP-Ethical-Analysis, SEP-Professional-Ethics, SEP-Privacy, SEP-Security)

Illustrative Learning Outcomes

CS Core:

1. Identify the five stages of the Data Life Cycle

DM-Core: Core Database System Concepts

CS Core:

1. Purpose and advantages of database systems
2. Components of database systems
3. Design of core DBMS functions (e.g., query mechanisms, transaction management, buffer management, access methods)
4. Database architecture, data independence, and data abstraction
5. Use of a declarative query language
6. Transaction mgmt
7. Normalization

8. Approaches for managing large volumes of data (e.g., noSQL database systems, use of MapReduce) (See also: PDC-Algorithms:2)
9. How to support CRUD-only applications
10. Distributed databases/cloud-based systems
11. Structured, semi-structured, and unstructured databases

KA Core:

12. Systems supporting structured and/or stream content

Illustrative Learning Outcomes
CS Core:

1. Identify at least four advantages that using a database system provides.
2. Enumerate the components of a (relational) database system.
3. Follow a query as it is processed by the components of a (relational) database system.
4. Defend the value of data independence.
5. Compose a simple select-project-join query in SQL.
6. Enumerate the four properties of a correct transaction manager.
7. Articulate the advantages for eliminating duplicate repeated data.
8. Outline how MapReduce uses parallelism to process data efficiently.
9. Evaluate the differences between structured and semi/unstructured databases.

DM-Modeling: Data Modeling

CS Core:

1. Data modeling (See also: SE-Requirements (Ask Titus))
2. Relational data model (See also: MSF-Discrete)

KA Core:

3. Conceptual models (e.g., entity-relationship, UML diagrams)
4. Semi-structured data models (expressed using DTD, XML, or JSON Schema, for example)

Non-Core:

5. Spreadsheet models
6. Object-oriented models (See also: FPL-OOP)
    a. GraphQL
7. New features in SQL
8. Specialized Data Modeling topics
    a. Time series data (aggregation, and join)
    b. Graph data (link traversal)
    c. Techniques for avoiding inefficient raw data access (e.g., “avg daily price”): materialized views and special data structures (e.g., Hyperloglog, bitmap)
    d. Geo-Spatial data (e.g., GIS databases) (See also: SPD-Access)

Illustrative Learning Outcomes
CS Core:

1. Articulate the components of the relational data model
2. Model 1:1, 1:n, and n:m relationships using the relational data model

KA Core:

3. Articulate the components of the E-R (or some other non-relational) data model.
4. Model a given environment using a conceptual data model.
5. Model a given environment using the document-based or key-value store-based data model.

DM-Relational: Relational Databases

CS Core:

1. Entity and referential integrity
    a. Candidate key, superkeys
2. Relational database design

KA Core:

3. Mapping conceptual schema to a relational schema
4. Physical database design: file and storage structures (See also OS-Files)
5. Introduction to Functional dependency Theory
6. Normalization Theory
    a. Decomposition of a schema; lossless-join and dependency-preservation properties of a decomposition
    b. Normal forms (BCNF)
    c. Denormalization (for efficiency)

Non-Core:

7. Functional dependency Theory
    a. Closure of a set of attributes
    b. Canonical Cover
8. Normalization Theory
    a. Multi-valued dependency (4NF)
    b. Join dependency (PJNF, 5NF)
    c. Representation theory

Illustrative Learning Outcomes

CS Core:

1. Articulate the defining characteristics behind the relational data model.
2. Comment on the difference between a foreign key and a superkey.
3. Enumerate the different types of integrity constraints.

KA Core:

4. Compose a relational schema from a conceptual schema which contains 1:1, 1:n, and n:m relationships.
5. Map appropriate file structure to relations and indices.

6. Articulate how functional dependency theory generalizes the notion of key.
7. Defend a given decomposition as lossless and or dependency preserving.
8. Detect which normal form a given decomposition yields.
9. Comment on reasons for denormalizing a relation.

DM-Querying: Query Construction

CS Core:

1. SQL Query Formation
    a. Interactive SQL execution
    b. Programmatic execution of an SQL query

KA Core:

2. Relational Algebra
3. SQL
    a. Data definition including integrity and other constraints specification
    b. Update sublanguage

Non-Core:

4. Relational Calculus
5. QBE and 4th-generation environments
6. Different ways to invoke non-procedural queries in conventional languages
7. Introduction to other major query languages (e.g., XPATH, SPARQL)
8. Stored procedures

Illustrative Learning Outcomes
CS Core:

1. Compose SQL queries that incorporate select, project, join, union, intersection, set difference, and set division.
2. Determine when a nested SQL query is correlated or not.
3. Iterate over data retrieved programmatically from a database via an SQL query.

KA Core:

4. Define, in SQL, a relation schema, including all integrity constraints and delete/update triggers.
5. Compose an SQL query to update a tuple in a relation.

DM-Processing: Query Processing

This knowledge area examines the process of how a Relational DBMS executes a query from its SQL origins, to its relational algebraic equivalence, to its optimized transformed equivalence, to a realized execution plan. This leads to the study of database tuning; the creation of specific indices (and denormalization) to allow for optimized query execution.

KA Core:

1. Page structures
2. Index structures
    a. B+ trees (See also: AL-Fundamentals)
    b. Hash indices: static and dynamic (See also: AL-Fundamentals, SEC-Foundations)
    c. Index creation in SQL
3. Algorithms for query operators
    a. External Sorting (See also: AL-Fundamentals)
    b. Selection
    c. Projection; with and without duplicate elimination
    d. Natural Joins: Nested loop, Sort-merge, Hash join
    e. Analysis of algorithm efficiency (See also: AL-Complexity)
4. Query transformations
5. Query optimization
    a. Access paths
    b. Query plan construction
    c. Selectivity estimation
    d. Index-only plans
6. Parallel Query Processing (e.g., parallel scan, parallel join, parallel aggregation) (See also: PDC-Algorithms)
7. Database tuning/performance
    a. Index selection
    b. Impact of indices on query performance (See also: SF-Performance:3, SEP-Sustainability)
    c. Denormalization

Illustrative Learning Outcomes

KA Core:

1. Articulate the purpose and organization of both B+ tree and hash index structures.
2. Compose an SQL query to create an index (any kind).
3. Articulate the steps for the various query operator algorithms: external sorting, projection with duplicate elimination, sort-merge join, hash-join, block nested-loop join.
4. Derive the run-time (in I/O requests) for each of the above algorithms
5. Transform a query in relational algebra to its equivalent appropriate for a left-deep, pipelined execution.
6. Compute selectivity estimates for a given selection and/or join operation.
7. Articulate how to modify an index structure to facilitate an index-only operation for a given relation.
8. For a given scenario decide on which indices to support for the efficient execution of a set of queries.
9. Articulate how DBMSs leverage parallelism to speed up query processing by dividing the work across multiple processors or nodes.

DM-Internals: DBMS Internals

This unit covers DBMS internals that are not directly involved in query execution.

KA Core:

1. DB Buffer Management (See also: SF-Resources)
2. Transaction Management (See also: PDC-Coordination:3)
    a. Isolation Levels
    b. ACID
    c. Serializability
    d. Distributed Transactions
3. Concurrency Control: (See also: OS-Concurrency)
    a. 2-Phase Locking
    b. Deadlocks handling strategies
    c. Quorum-based consistency models
4. Recovery Manager
    a. Relation with Buffer Manager

Non-Core:

5. Concurrency Control:
    a. Optimistic CC
    b. Timestamp CC
6. Recovery Manager
    a. Write-Ahead logging
    b. ARIES recovery system (Analysis, REDO, UNDO)

Illustrative Learning Outcomes

KA Core:

1. Articulate how a DBMS manages its Buffer Pool
2. Articulate the four properties for a correct transaction manager
3. Outline the principle of serializability

DM-NoSQL: NoSQL Systems

KA Core:

1. Why NoSQL? (e.g., Impedance mismatch between Application [CRUD] and RDBMS)
2. Key-Value and Document data model

Non-Core:

3. Storage systems (e.g., Key-Value systems, Data Lakes)
4. Distribution Models (Sharding and Replication) (See also: PDC-Communication:4)
5. Graph Databases
6. Consistency Models (Update and Read, Quorum consistency, CAP theorem) (See also: PDC-Communication:4)
7. Processing model (e.g., Map-Reduce, multi-stage map-reduce, incremental map-reduce) (See also: PDC-Communication:4)

8. Case Studies: Cloud storage system (e.g., S3); Graph databases ; “When not to use NoSQL” (See also: SPD-Web: 7)

Illustrative Learning Outcomes
KA Core:

1. Articulate a use case for the use of NoSQL over RDBMS.
2. Articulate the defining characteristics behind Key-Value and Document-based data models.

DM-Security: Data Security and Privacy

KA Core:

1. Need for, and different approaches to securing data at rest, in transit, and during processing.
2. Protecting data and database systems from attacks, including injection attacks such as SQL injection (See also: SEC-Security)
3. Database auditing and its role in digital forensics
4. Differences between data security and data privacy
5. Personally identifying information (PII) and its protection
6. Data inferencing and preventing attacks
7. Laws and regulations governing data security and data privacy (See also: SEP-???)
8. Ethical considerations in ensuring the security and privacy of data (See also: SEP-???)

Non-Core:

9. Typical risk factors and prevention measures for ensuring data integrity
10. Ransomware and prevention of data loss and destruction

Illustrative Learning Outcomes
KA Core:

1. Apply several data exploration approaches to understanding unfamiliar datasets.
2. Identify and mitigate risks associated with different approaches to protecting data.
3. Describe the differences in the goals for data security and data privacy.
4. Develop a database auditing system given risk considerations.
5. Describe legal and ethical considerations of end-to-end data security and privacy.

DM-Analytics: Data Analytics

KA Core:

1. Exploratory data techniques (e.g., Raj to fill in)
2. Data science lifecycle: business understanding, data understanding, data preparation, modeling, evaluation, deployment, and user acceptance
3. Data mining and machine learning algorithms: e.g., classification, clustering, association, regression (See also: AI-ML)

4. Data acquisition and governance
5. Data security and privacy considerations (See also: SEP-Security)
6. Data fairness and bias (See also: SEP-Security, AI-Ethics)
7. Data visualization techniques and their use in data analytics (Git-Ask Susan)
8. Entity Resolution

Illustrative Learning Outcomes
KA Core:
1. Describe several data exploration approaches, including visualization, to understanding unfamiliar datasets.
2. Apply several data exploration approaches to understanding unfamiliar datasets.
3. Describe basic machine learning/data mining algorithms and when they are appropriate for use.
4. Apply several machine learning/data mining algorithms.
5. Describe legal and ethical considerations in acquiring, using, and modifying datasets.
6. Describe issues of fairness and bias in data collection and usage

DM-Distributed: Distributed Databases/Cloud Computing

Non-Core:
1. Distributed DBMS (PDC-Communications)
    a. Distributed data storage
    b. Distributed query processing
    c. Distributed transaction model
    d. Homogeneous and heterogeneous solutions
    e. Client-server distributed databases (See also: NC-Introduction)
2. Parallel DBMS (See also: PDC-Algorithms)
    a. Parallel DBMS architectures: shared memory, shared disk, shared nothing;
    b. Speedup and scale-up, e.g., use of the MapReduce processing model (cross-reference CN/Processing, PD/Parallel Decomposition) (See also: SF-Basics:8)
    c. Data replication and weak consistency models (See also: PDC-Coordination)

DM-Unstructured: Semi-structured and Unstructured Databases

Non-Core:
1. Vectorized unstructured data (text, video, audio, etc.) and vector storage
    a. TF-IDF Vectorizer with ngram
    b. Word2Vec
    c. Array database or array data type handling
2. Semi-structured (e.g., JSON)
    a. Storage
        i. Encoding and compression of nested data types
    b. Indexing
        i. Btree, skip index, Bloom filter

        ii. Inverted index and bitmap compression
        iii. Space filling curve indexing for semi-structured geo-data
    c. Query processing for OLTP and OLAP use cases
        i. Insert, Select, update/delete trade offs
        ii. Case studies on Postgres/JSON, MongoDB and Snowflake/JSON

DM-SEP: Society, Ethics, and Professionalism

1. Issues related to scale (See also SEP-Economies)
2. Data privacy overall (See also: SEP-Privacy, SEP-Ethical-Analysis)
    a. Privacy compliance by design (See also: Privacy)
3. Data anonymity (See also: SEP-Privacy)
4. Data ownership/custodianship (See also: SEP-Professional-Ethics)
5. Reliability of data (See also: SEP-Security)
6. Intended and unintended applications of stored data (See also: SEP-Professional-Ethics)
7. Provenance, data lineage, and metadata management (See also: SEP-Professional-Ethics)
8. Data security (See also: SEP-Security)

Illustrative Learning Outcomes

1. Enumerate three social and three legal issues related to large data collections
2. Articulate the value of data privacy
3. Identify the competing stakeholders with respect to data ownership
4. Enumerate three negative unintended consequences from a given (well known) data-centric application (e.g., Facebook, LastPass, Ashley Madison)

Professional Dispositions

- **Meticulous:** Those who either access or store data collections must be meticulous in fulfilling data ownership responsibilities.
- **Responsible**: In conjunction with the professional management of (personal) data, it is equally important that data be managed responsibly. Protection from unauthorized access as well as prevention of irresponsible, though legal, use of data is paramount. Furthermore, data custodians need to protect data not only from outside attack, but from crashes and other foreseeable dangers.
- **Collaborative**: Data managers and data users must behave in a collaborative fashion to ensure that the correct data is accessed, and is used only in an appropriate manner.
- **Responsive**: The data that gets stored and is accessed is always in response to an institutional need/request.

Math Requirements

Required:

- Discrete Mathematics

  - Set theory (union, intersection, difference, cross-product)

Desirable Data Structures:

- Hash functions and tables
- Balanced (binary) trees (e.g., AVL, 2-3-4, Red-Black)

Course Packaging Suggestions

For those implementing a single course on Database Systems, there are a variety of options. As described in [1], there are four primary perspectives from which to approach databases:

- Database design/modeling
- Database use
- Database administration
- Database development, which includes implementation algorithms

Course design proceeds by focusing on topics from each perspective in varying degrees according to one's institutional context. For example in [1], one of the courses described can be characterized as design/modeling (20%), use (20%), development/internals (30%), and administration/tuning/advanced topics (30%). The topics might include:

- DM-SEP: Society, Ethics, and Professionalism (3 hours)
- DM-Data: The Role of Data (1 hour)
- DM-Core: Core Database System Concepts (3 hours)
- DM-Modeling: Data Modeling (5 hours)
- DM-Relational: Relational Databases (4 hours)
- DM-Querying: Query Construction (6 hours)
- DM-Processing: Query Processing (5 hours)
- DM-Internals: DBMS Internals (5 hours)
- DM-NoSQL: NoSQL Systems (4 hours)
- DM-Security: Data Security and Privacy (3 hours)
- DM-Distributed: Distributed Databases/Cloud Computing (2 hours)

Possibly, the more interesting question is how to cover the CS Core concepts in the absence of a dedicated database course. Perhaps the key to accomplishing this is to *normalize* database access. Starting with the introductory course students could be accessing a database versus file I/O or interactive data entry, to acquire the data needed for introductory-level programming. As students progress through their curriculum, additional CS Core topics can be introduced. For example, introductory students would be given the code to access the database along with the SQL query. By the intermediate level, they could be writing their own queries. Finally, in a Software Engineering or capstone course, they are practicing database design. One advantage of this *databases across the curriculum* approach is that allows for the inclusion of database-related SEP topics to also be spread across the curriculum.

In a similar vein one might have a whole course on the Role of Data from either a Security (SEC) perspective, or and Ethics (SEP) Perspective.

[1] The 2022 Undergraduate Database Course in Computer Science: What to Teach?. Michael Goldweber, Min Wei, Sherif Aly, Rajendra K. Raj, and Mohamed Mokbel. ACM Inroads, Volume 13, Number 3, 2022.

Committee

Chair: Mikey Goldweber, Denison University, Granville, OH, USA

Members:

- Sherif Aly, The American University in Cairo, Cairo, Egypt
- Sara More, Johns Hopkins University, Baltimore, MD, USA
- Mohamed Mokbel, University of Minnesota, Minneapolis, MN, USA
- Rajendra Raj, Rochester Institute of Technology, Rochester, NY, USA
- Avi Silberschatz, Yale University, New Haven, CT, USA
- Min Wei, Seattle, WA, Microsoft
- Qiao Xiang, Xiamen University, Xiamen, China

Foundations of Programming Languages (FPL)

Preamble

This knowledge area provides a basis (rooted in discrete mathematics and logic) for the understanding of complex modern programming languages: their foundations, implementation, and formal description. Although programming languages vary according to the language paradigm and the problem domain and evolve in response to both societal needs and technological advancement, they share an underlying abstract model of computation and program development. This remains true even as processor hardware and their interface with programming tools are becoming increasingly intertwined and progressively more complex. An understanding of the common abstractions and programming paradigms enables faster learning of new programming languages.

The Foundations of Programming Languages Knowledge Area is concerned with articulating the underlying concepts and principles of programming languages, the formal specification of a programming language and the behavior of a program, explaining how programming languages are implemented, comparing the strengths and weaknesses of various programming paradigms, and describing how programming languages interface with entities such as operating systems and hardware. The concepts covered in this area are applicable to many different languages and an understanding of these principles assists in being able to move readily from one language to the next, and to be able to select a programming paradigm and a programming language to best suit the problem at hand.

Two example courses are presented at the end of this knowledge area to illustrate how the content may be covered. The first is an introductory course which covers the CS Core and KA Core content. This is a course focused on the different programming paradigms and ensue familiarity with each to a level sufficient to be able to decide which paradigm is appropriate in which circumstances.

The second course is an advanced course focused on the implementation of a programming language and the formal description of a programming language and a formal description of the behavior of a program.

While these two courses have been the predominant way to cover this knowledge area of the past decade, it is by no means the only way that the content can be covered. An institution could, for example, choose to cover only the CS Core content (28 hours) in a shorter course, or in a course which combines this CS Core content with Core content from another knowledge area such as Software Engineering. Natural combinations are easily identifiable since they are the areas in which the Foundations of Programming Languages knowledge areas overlaps with other knowledge areas. A list of such overlap areas is provided at the end of this knowledge area.

Programming languages are the medium through which programmers precisely describe concepts, formulate algorithms, and reason about solutions. Over the course of a career, a computer scientist will need to learn and work with many different languages, separately or together. Software developers must understand the programming models, new programming features and constructs, 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 to improve execution efficiency and long-term maintenance of developed software. The effective use of programming languages and appreciation of their limitations also requires a basic knowledge of programming language translation and program analysis, of run-time behavior, and components such as memory management and the interplay of concurrent processes communicating with each other through message-passing, shared memory, and synchronization, Finally, some developers and researchers will need to design new languages, an exercise which requires familiarity with basic principles.

Changes since CS 2013

These include a change in name of the Knowledge Area from Programming Languages to Foundations of Programming Languages to better reflect the fact that the KA is about the fundamentals underpinning programming languages and related concepts, and not about any specific programming languages. Changes also include a redistribution of content formerly identified as core tier-1 and core tier-2 within the Programming Language Knowledge Area (KA). In CS2013, graduates were expected to complete all tier-1 hours and 80% of the tier-2 hours, a total of 24 required hours. These are now CS Core hours. The remaining tier-2 hours are now KA Core hours. All computer science graduates are expected to have the CS Core hours, and those graduates that specialize in a knowledge area are also expected to have the majority of the KA core hours. Content that is not identified as either CS Core hours or KA Core hours are non-core topics. The variation in core hours (tier-1 plus 80% of tier-2 hours) from 2013 reflects the change in importance or relevance of topics over the past decade. The inclusion of new topics is driven by their current prominence in the programming language landscape, or the anticipated impact of emerging areas on the profession in general. Specifically, the changes are:

● Object-Oriented Programming	-3 CS Core hours
● Functional Programming	-2 CS Core hours
● Event-Driven and Reactive Programming	+1 CS Core hour
● Parallel and Distributed Computing	+2 CS Core hours
● Type Systems	-1 CS Core hour
● Language Translation and Execution	-3 CS Core hours

In addition, some knowledge units from CS 2013 are renamed to more accurately reflect their content:

- Static Analysis is renamed to Program Analysis and Analyzers
- Concurrency and Parallelism is renamed to Parallel and Distributed Computing
- Program Representation is renamed to Program Abstraction and Representation
- Runtime Systems is renamed to Runtime Behavior and Systems
- Basic Type Systems and Type Systems were merged into a single topic and named Type Systems

Seven new knowledge units have been added to reflect their continuing and growing importance as we look toward the 2030s:

● Compiler vs Interpreted Languages	+1 CS Core hour
● Scripting	+2 CS Core hours

- Systems Execution and Memory Model +3 CS Core hours
- Formal Development Methodologies
- Design Principles of Programming Languages
- Quantum Computing
- Fundamentals of Programming Languages and Society, Ethics and Professionalism

Compared to CS 2013 which had a total of 24 CS core hours (tier-1 hours plus 80% of tier-2 hours), and 4 KA core hours (20% of tier-2 hours), the current recommendation has a total of 22 CS core hours and 20 KA core hours. Note that there is no requirement that each computer science graduate sees any KA core hours – they are a recommendation of content to consider if a program choses to offer greater emphasis on this knowledge area.

Note:

- Several topics within this knowledge area either build on, or overlap with, content covered in other knowledge areas such as 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.
- Different programming paradigms correspond to different problem domains. Most languages have evolved to integrate more than one programming paradigms such imperative with OOP, functional programming with OOP, logic programming with OOP, and event and reactive modeling with OOP. Hence, the emphasis is not on just one programming paradigm but a balance of all major programming paradigms.
- While the number of CS core and KA core hours is identified for each major programming paradigm (object-oriented, functional, logic), the distribution of hours across the paradigms may differ depending on the curriculum and programming languages students have been exposed to leading up to coverage of this knowledge area. This document makes the assumption that students have exposure to a object-oriented programming language leading into this knowledge area.
- Imperative programming is not listed as a separate paradigm to be examined, instead it is treated as a subset of the object-oriented paradigm.
- With multicore computing, cloud computing, and computer networking becoming commonly available in the market, it has become imperative to understand the integration of "Distribution, concurrency, parallelism" along with other programming paradigms as a core area. This paradigm is integrated with almost all other major programming paradigms.
- With ubiquitous computing and real-time temporal computing becoming more in daily human life such as health, transportation, smart homes, it has become important to cover the software development aspects of event-driven and reactive programming, as well as parallel and distributed computing, under the programming languages knowledge area. Some of the topics covered will require, and interface with, concepts covered in knowledge areas such as Architecture and Organization, Operating Systems, and Systems Fundamentals.
- Some topics from the Parallel and Distributed Computing Knowledge Unit are likely to be integrated within the curriculum with topics from the Parallel and Distributed Programming Knowledge Area.

- There is an increasing interest in formal methods to prove program correctness and other properties. To support this, increased coverage of topics related to formal methods is included, but all of these topics are identified as non-core.
- When introducing these topics, it is also important that an instructor provides context for this material including why we have an interest in programming languages, and what they do for us in terms of providing a human readable version of instructions to a computer to execute for us.

Core Hours

Knowledge Units	**CS Core**	**KA Core**
Object-Oriented Programming	5	1
Functional Programming	4	3
Logic Programming		3
Compiled vs Interpreted Languages	1	
Scripting	2	
Event-Driven and Reactive Programming	2	2
Parallel and Distributed Computing	3	2
Type Systems	3	4
Systems Execution and Memory Model	3	
Language Translation and Execution		2
Program Abstraction and Representation		3
Syntax Analysis		
Compiler Semantic Analysis		
Program Analysis and Analyzers		
Code Generation		
Runtime Behavior and Systems		
Advanced Programming Constructs		
Language Pragmatics		
Formal Semantics		
Formal Development Methodologies		
Design Principles of Programming Languages		
Quantum Computing		
FPL and SEP		
Total	**23**	**20**

Knowledge Units

FPL-OOP: Object-Oriented Programming

CS Core:

1. Imperative programming as a sunset if object-oriented programming
2. Object-oriented design

    a. Decomposition into objects carrying state and having behavior
    b. Class-hierarchy design for modeling
3. Definition of classes: fields, methods, and constructors (See also: SDF-Fundamentals)
4. Subclasses, inheritance (including multiple inheritance), and method overriding
5. Dynamic dispatch: definition of method-call
6. Exception handling (See also: PDC-Coordination, SF-System-Reliability)
7. Object-oriented idioms for encapsulation
    a. Privacy, data hiding, and visibility of class members
    b. Interfaces revealing only method signatures
    c. Abstract base classes, traits and mixins
8. Dynamic vs static properties
9. Composition vs inheritance
10. Subtyping
    a. Subtype polymorphism; implicit upcasts in typed languages
    b. Notion of behavioral replacement: subtypes acting like supertypes
    c. Relationship between subtyping and inheritance

KA Core:

11. Collection classes, iterators, and other common library components

Illustrative Learning Outcomes:

CS Core:

1. Enumerate the difference between the imperative and object-oriented programming paradigms.
2. Compose a class through design, implementation, and testing to meet behavioral requirements.
3. Build a simple class hierarchy utilizing subclassing that allows code to be reused for distinct subclasses.
4. Predict and validate control flow in a program using dynamic dispatch.
5. Compare and contrast:
    c. the procedural/functional approach-defining a function for each operation with the function body providing a case for each data variant-and
    d. the object-oriented approach-defining a class for each data variant with the class definition providing a method for each operation.
    e. Understand both as defining a matrix of operations and variants. (See also: FPL-Functional)
6. Compare and contrast the benefits and costs/impact of using inheritance (subclasses) and composition (in particular how to base composition on higher order functions).
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).
8. Use object-oriented encapsulation mechanisms such as interfaces and private members.
9. Define and use iterators and other operations on aggregates, including operations that take functions as arguments, in multiple programming languages, selecting the most natural idioms for each language. (See also: FPL-Functional)

KA Core:

10. Use collection classes and iterators effectively to solve a problem.

FPL-Functional: Functional Programming

CS Core:

1. Lambda expressions and evaluation (See also: AL-Models, FPL-Formalism)
    a. Variable binding and scope rules (See also: SDF-Fundamentals)
    b. Parameter passing (See also: SDF-Fundamentals)
    c. Nested lambda expressions and reduction order
2. Effect-free programming
    a. Function calls have no side effects, facilitating compositional reasoning
    b. Immutable variables and data copying vs. reduction
    c. Use of recursion vs. loops vs. pipelining (map/reduce)
3. Processing structured data (e.g., trees) via functions with cases for each data variant
    a. Functions defined over compound data in terms of functions applied to the constituent pieces
    b. Persistent data structures
4. Using higher-order functions (taking, returning, and storing functions)

KA Core:

5. Function closures (functions using variables in the enclosing lexical environment)
    a. Basic meaning and definition - creating closures at run-time by capturing the environment
    b. Canonical idioms: call-backs, arguments to iterators, reusable code via function arguments
    c. Using a closure to encapsulate data in its environment
    d. Lazy versus eager evaluation

Non-Core:

6. Graph reduction machine and call-by-need
7. Implementing lazy evaluation
8. Integration with logic programming paradigm using concepts such as equational logic, narrowing, residuation and semantic unification (See also: FPL-Logic)
9. Integration with other programming paradigms such as imperative and object-oriented

Illustrative learning outcomes:

CS Core:

1. Develop basic algorithms that avoid assigning to mutable state or considering reference equality.
2. Develop useful functions that take and return other functions.
3. Compare and contrast:
    a. the procedural/functional approach-defining a function for each operation with the function body providing a case for each data variant, and
    b. 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. (See also: FPL-OOP)

KA Core:

4. Explain a simple example of lambda expression being implemented using a virtual machine, such as a SECD machine, showing storage and reclaim of the environment.
5. Correctly interpret variables and lexical scope in a program using function closures.
6. Use functional encapsulation mechanisms such as closures and modular interfaces.
7. Compare and contrast stateful vs stateless execution.
8. Define and use iterators and other operations on aggregates, including operations that take functions as arguments, in multiple programming languages, selecting the most natural idioms for each language. (See also: FPL-OOP)

Non-Core:

9. Illustrate graph reduction using a λ-expression using a shared subexpression
10. Illustrate the execution of a simple nested λ-expression using an abstract machine, such as an ABC machine.
11. Illustrate narrowing, residuation and semantic unification using simple illustrative examples.
12. Illustrate the concurrency constructs using simple programming examples of known concepts such as a buffer being read and written concurrently or sequentially. (See also: FPL-OOP)

FPL-Logic: Logic Programming

KA Core:

1. Universal vs. existential quantifiers (See also: AI-LRR, MSF-Discrete-mathematics)
2. First order predicate logic vs. higher order logic (See also: AI-LRR, MSF-Discrete-mathematics:8)
3. Expressing complex relations using logical connectives and simpler relations
4. Definitions of Horn clause, facts, goals and subgoals
5. Unification and unification algorithm; unification vs. assertion vs expression evaluation
6. Mixing relations with functions {see also: MSF-Discrete-mathematics:1)
7. Cuts, backtracking and non-determinism
8. Closed-world vs. open-world assumptions

Non-Core:

9. Memory overhead of variable copying in handling iterative programs
10. Programming constructs to store partial computation and pruning search trees
11. Mixing functional programming and logic programming using concepts such as equational logic, narrowing, residuation and semantic unification (See also: FPL-Functional)
12. Higher-order, constraint and inductive logic programming (See also: AI-LRR)
13. Integration with other programming paradigms such as object-oriented programming
14. Advance programming constructs such as difference-lists, creating user defined data structures, set of, etc.

Illustrative learning outcomes:
KA Core:

1. Use a logic language to implement a conventional algorithm.
2. Use a logic language to implement an algorithm employing implicit search using clauses, relations, and cuts.
3. Use a simple illustrative example to show correspondence between First Order Predicate Logic (FOPL) and logic programs using Horn clauses.
4. Use examples to illustrate unification algorithm and its role of parameter passing in query reduction.
5. Use simple logic programs interleaving relations, functions, and recursive programming such as factorial and Fibonacci numbers and simple complex relationships between entities, and illustrate execution and parameter passing using unification and backtracking.

Non-Core:

6. Illustrate computation of simple programs such as Fibonacci and show overhead of recomputation, and then show how to improve execution overhead.

FPL-Compile-Interpret: Compiled vs Interpreted Languages Programming

CS Core:

1. Execution models for compiled languages and interpreted languages
2. Use of intermediate code, e.g., Bytecode
3. Limitations and benefits of compiled and interpreted languages

Illustrative Learning Outcomes:
CS Core:

1. Explain and understand the differences between compiled and interpreted languages, including the benefits and limitations of each.

FPL-Scripting: Scripting

CS Core:

1. Error/exception handling
2. Piping (See also: AR-Organization, SF-Overview:6, OS-Process:5)
3. System commands (See also: SF-A)
    a. Interface with operating systems (See also: SF-Overview, OS-Principles:3)
4. Environment variables (See also: SF-A)
5. File abstraction and operators (See also: SDF-Fundamentals, OS-Files, AR-IO, SF-Resource)
6. Data structures, such as arrays and lists (See also: AL-Fundamentals, SDF-Fundamentals, SDF-DataStructures)
7. Regular expressions (See also: AL-Models)
8. Programs and processes (See also: OS-Process)
9. Workflow

Illustrative learning outcomes:

CS Core:

1. Create and execute automated scripts to manage various system tasks.
2. Solve various text processing problems through scripting

FPL-Event-Driven: Event-Driven and Reactive Programming

CS Core:

1. Procedural programming vs. reactive programming: advantages of reactive programming in capturing events
2. Components of reactive programming: event-source, event signals, listeners and dispatchers, event objects, adapters, event-handlers (See also: GIT-Interaction, SPD-Web, SPD-Mobile, SPD-Robot, SPD-Embedded, SPD-Game, SPD-Interactive)
3. Stateless and state-transition models of event-based programming
4. Canonical uses such as GUIs, mobile devices, robots, servers (See also: GIT-Interaction, GIT-Image Processing, SPD-Web, SPD-Mobile, SPD-Robot, SPD-Embedded, SPD-Game, SPD-Interactive)

KA Core:

5. Using a reactive framework
    a. Defining event handlers/listeners
    b. Parameterization of event senders and event arguments
    c. Externally-generated events and program-generated events
6. Separation of model, view, and controller

Illustrative learning outcomes:

CS Core:

1. Implement event handlers for use in reactive systems, such as GUIs.
2. Examine why an event-driven programming style is natural in domains where programs react to external events.

KA Core:

3. Define and use a reactive framework.
4. Describe an interactive system in terms of a model, a view, and a controller.

FPL-Parallel: Parallel and Distributed Programming

CS Core:

1. Safety and liveness (See also: PDC-Evaluation)
    a. Race conditions (See also: OS-Concurrency:2)
    b. Dependencies/preconditions
    c. Fault models (OS-Faults)

    d. Termination {see also: PDC-Coordination)

2. Programming models (See also: PDC-Programs).
        a. One or more of the following:
    a. Actor models
    b. Procedural and reactive models
    c. Synchronous/asynchronous programming models
    d. Data parallelism
3. Properties {see also: PDC-Programs, PDC-Coordination)
    a. Order-based properties:
        i. Commutativity
        ii. Independence
    b. Consistency-based properties:
        i. Atomicity
        ii. Consensus
4. Execution control (See also: PDC-Coordination, SF-B)
    a. Async await
    b. Promises
    c. Threads
5. Communication and coordination (See also: OS-Process:5, PDC-Communication, PDC-Coordination)
    a. Message-passing
    b. Shared memory
    c. cobegin-coend
    d. Monitors
    e. Channels
    f. Threads
    g. Guards

KA Core:

6. Futures
7. Language support for data parallelism such as forall, loop unrolling, map/reduce
8. Effect of memory-consistency models on language semantics and correct code generation
9. Representational State Transfer Application Programming Interfaces (REST APIs)
10. Technologies and approaches: cloud computing, high performance computing, quantum computing, ubiquitous computing
11. Overheads of message passing
12. Granularity of program for efficient exploitation of concurrency.
13. Concurrency and other programming paradigms (e.g., functional).

Illustrative learning outcomes:

CS Core:

1. Explain why programming languages do not guarantee sequential consistency in the presence of data races and what programmers must do as a result.
2. Implement correct concurrent programs using multiple programming models, such as shared memory, actors, futures, synchronization constructs, and data-parallelism primitives.
3. \Use a message-passing model to analyze a communication protocol.
4. Use synchronization constructions such as monitor/synchronized methods in a simple program.
5. Modeling data dependency using simple programming constructs involving variables, read and write.
6. Modeling control dependency using simple constructs such as selection and iteration.

KA Core:

7. Explain how REST API's integrate applications and automate processes.
8. Explain benefits, constraints and challenges related to distributed and parallel computing.

FPL-Types: Type Systems

CS Core:

1. A type as a set of values together with a set of operations
   a. Primitive types (e.g., numbers, Booleans) (See also: SDF-Fundamentals)
   b. Compound types built from other types (e.g., records, unions, arrays, lists, functions, references using set operations) (See also: SDF-DataStructures)
2. Association of types to variables, arguments, results, and fields
3. Type safety as an aspect of program correctness (See also: FPL-Formalism)
4. Type safety and errors caused by using values inconsistently given their intended types
5. Goals and limitations of static and dynamic typing
   a. Detecting and eliminating errors as early as possible
6. Generic types (parametric polymorphism)
   a. Definition and advantages of polymorphism: parametric, subtyping, overloading and coercion
   b. Comparison of monomorphic and polymorphic types
   c. Comparison with ad-hoc polymorphism (overloading) and subtype polymorphism
   d. Generic parameters and typing
   e. Use of generic libraries such as collections
   f. Comparison with ad hoc polymorphism (overloading) and subtype polymorphism
   g. Prescriptive vs. descriptive polymorphism
   h. Implementation models of polymorphic types
   i. Subtyping

KA Core:

7. Type equivalence: structural vs name equivalence
8. Complementary benefits of static and dynamic typing
   a. Errors early vs. errors late/avoided
   b. Enforce invariants during code development and code maintenance vs. postpone typing decisions while prototyping and conveniently allow flexible coding patterns such as heterogeneous collections

   c. Typing rules
      i. Rules for function, product, and sum types
   d. Avoid misuse of code vs. allow more code reuse
   e. Detect incomplete programs vs. allow incomplete programs to run
   f. Relationship to static analysis
   g. Decidability

Non-Core:

9. Compositional type constructors, such as product types (for aggregates), sum types (for unions), function types, quantified types, and recursive types
10. Type checking
11. Subtyping (See also: FPL-OOP)
    a. Subtype polymorphism; implicit upcasts in typed languages
    b. Notion of behavioral replacement: subtypes acting like supertypes
    c. Relationship between subtyping and inheritance
12. Type safety as preservation plus progress
13. Type inference
14. Static overloading
15. Propositions as types (implication as a function, conjunction as a product, disjunction as a sum) (See also: FPL-Formalism)
16. Dependent types (universal quantification as dependent function, existential quantification as dependent product) (See also: FPL-Formalism)

Illustrative learning outcomes:
CS Core:

1. Describe, for both a primitive and a compound type, the values that have that type.
2. Describe, for a language with a static type system, the operations that are forbidden statically, such as passing the wrong type of value to a function or method.
3. Describe examples of program errors detected by a type system.
4. Identify program properties, for multiple programming languages, that are checked statically and program properties that are checked dynamically.
5. Describe an example program that does not type-check in a particular language and yet would have no error if run.
6. Use types and type-error messages to write and debug programs.

KA Core:

7. Explain how typing rules define the set of operations that are legal for a type.
8. List the type rules governing the use of a particular compound type.
9. Explain why undecidability requires type systems to conservatively approximate program behavior.
10. Define and use program pieces (such as functions, classes, methods) that use generic types, including for collections.
11. Discuss the differences among generics, subtyping, and overloading.

12. Explain multiple benefits and limitations of static typing in writing, maintaining, and debugging software.

Non-Core:

13. Define a type system precisely and compositionally.
14. For various foundational type constructors, identify the values they describe and the invariants they enforce.
15. Precisely describe the invariants preserved by a sound type system.
16. Prove type safety for a simple language in terms of preservation and progress theorems.
17. Implement a unification-based type-inference algorithm for a simple language.
18. Explain how static overloading and associated resolution algorithms influence the dynamic behavior of programs.

FPL-Systems: Systems Execution and Memory Model

CS Core:

1. Data structures for translation, execution, translation and code mobility such as stack, heap, aliasing (sharing using pointers), indexed sequence and string
2. Direct, indirect, and indexed access to memory location
3. Run-time representation of data abstractions such as variables, arrays, vectors, records, pointer-based data elements such as linked-lists and trees, and objects
4. Abstract low-level machine with simple instruction, stack and heap to explain translation and execution
5. Run-time layout of memory: activation record (with various pointers), static data, call-stack, heap (See also: AR-Memory, OS-Memory)
    a. Translating selection and iterative constructs to control-flow diagrams
    b. Translating control-flow diagrams to low level abstract code
    c. Implementing loops, recursion, and tail calls
    d. Translating function/procedure calls and return from calls, including different parameter passing mechanism using an abstract machine
6. Memory management (See also: AR-Memory, OS-Memory)
    a. Low level allocation and accessing of high-level data structures such as basic data types, n-dimensional array, vector, record, and objects
    b. Return from procedure as automatic deallocation mechanism for local data elements in the stack
    c. Manual memory management: allocating, de-allocating, and reusing heap memory
    d. Automated memory management: garbage collection as an automated technique using the notion of reachability
7. Green computing (See also: SEP-Sustainability)

Illustrative learning outcomes:

CS Core:

1. Diagram a low-level run-time representation of core language constructs, such as data abstractions and control abstractions.

2. 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.
3. Investigate, identify, and fix memory leaks and dangling-pointer dereferences.

FPL-Translation: Language Translation and Execution

CS Core:

1. Interpretation vs. compilation to native code vs. compilation to portable intermediate representation
    a. BNF and extended BNF representation of context-free grammar
    b. Parse tree using a simple sentence such as arithmetic expression or if-then-else statement
    c. Execution as native code or within a virtual machine
2. Language translation pipeline: syntax analysis, parsing, optional type-checking, translation/code generation and optimization, linking, loading, execution

KA Core:

3. Run-time representation of core language constructs such as objects (method tables) and first-class functions (closures)
4. Secure compiler development (See also: SEC-Foundation, SEC-Defense)

Illustrative learning outcomes:
CS Core:

1. Differentiate a language definition (what constructs mean) from a particular language implementation (compiler vs. interpreter, run-time representation of data objects, etc.).
2. Differentiate syntax and parsing from semantics and evaluation.
3. Use BNF and extended BNF to specify the syntax of simple constructs such as if-then-else, type declaration and iterative constructs for known languages such as C++ or Python.
4. Illustrate parse tree using a simple sentence/arithmetic expression.
5. Illustrate translation of syntax diagrams to BNF/extended BNF for simple constructs such as if-then-else, type declaration, iterative constructs, etc.
6. Illustrate ambiguity in parsing using nested if-then-else/arithmetic expression and show resolution using precedence order

Non-Core:

7. Discuss the benefits and limitations of garbage collection, including the notion of reachability.

FPL-Abstraction: Program Abstraction and Representation

KA Core:

1. BNF and regular expressions
2. Programs that take (other) programs as input such as interpreters, compilers, type-checkers, documentation generators

3. Components of a language
    a. Definitions of alphabets, delimiters, sentences, syntax and semantics
    b. Syntax vs. semantics
4. Program as a set of non-ambiguous meaningful sentences
5. Basic programming abstractions: constants, variables, declarations (including nested declarations), command, expression, assignment, selection, definite and indefinite iteration, iterators, function, procedure, modules, exception handling (See also: SDF-Fundamental)
6. Mutable vs. immutable variables: advantages and disadvantages of reusing existing memory location vs. advantages of copying and keeping old values; storing partial computation vs. recomputation
7. Types of variables: static, local, nonlocal, global; need and issues with nonlocal and global variables
8. Scope rules: static vs. dynamic; visibility of variables; side-effects
9. Side-effects induced by nonlocal variables, global variables and aliased variables

Non-Core:

10. L-values and R-values: mapping mutable variable-name to L-values; mapping immutable variable-names to R-values (See also: SDF-A)
11. Environment vs. store and their properties
12. Data and control abstraction
13. Mechanisms for information exchange between program units such as procedures, functions and modules: nonlocal variables, global variables, parameter passing, import-export between modules
14. Data structures to represent code for execution, translation, or transmission
15. Low level instruction representation such as virtual machine instructions, assembly language, and binary representation (See also: AR-B, AR-C)
16. Lambda calculus, variable binding, and variable renaming (See also: AL-Models, FPL-Formalism)
17. Types of semantics: operational, axiomatic, denotational, behavioral; define and use abstract syntax trees; contrast with concrete syntax

Illustrative learning outcomes:
KA Core:

1. Illustrate the scope of variables and visibility using simple programs
2. Illustrate different types of parameter passing using simple pseudo programming language
3. Explain side-effect using global and nonlocal variables and how to fix such programs
4. Explain how programs that process other programs treat the other programs as their input data.
5. Describe a grammar and an abstract syntax tree for a small language.
6. Describe the benefits of having program representations other than strings of source code.
7. Implement a program to process some representation of code for some purpose, such as an interpreter, an expression optimizer, or a documentation generator.

FPL-Syntax: Syntax Analysis

Non-Core:

1. Regular grammars vs. context-free grammars (See also: AL-Models)

2. Scanning and parsing based on language specifications
3. Lexical analysis using regular expressions
4. Tokens and their use
5. Parsing strategies including top-down (e.g., recursive descent, or LL) and bottom-up (e.g., LR or GLR) techniques.
    a. Lookahead tables and their application to parsing
6. Language theory
    a. Chomsky hierarchy (See also: AL-Models)
    b. Left-most/right-most derivation and ambiguity
    c. Grammar transformation
7. Parser error recovery mechanisms
8. Generating scanners and parsers from declarative specifications

Illustrative learning outcomes:
Non-Core:

1. Use formal grammars to specify the syntax of languages.
2. Illustrate the role of lookahead tables in parsing.
3. Use declarative tools to generate parsers and scanners.
4. Recognize key issues in syntax definitions: ambiguity, associativity, precedence.

FPL-Semantics: Compiler Semantic Analysis

Non-Core:

1. Abstract syntax trees; contrast with concrete syntax
2. Defining, traversing and modifying high-level program representations
3. Scope and binding resolution
4. Static semantics
    a. Type checking
    b. Define before use
    c. Annotation and extended static checking frameworks
5. L-values/R-values (See also: SDF-Fundamentals)
6. Call semantics
7. Types of parameter-passing with simple illustrations and comparison: call by value, call by reference, call by value-result, call by name, call by need and their variations
8. Declarative specifications such as attribute grammars and their applications in handling limited context-base grammar

Illustrative learning outcomes:
Non-Core:

1. Describe an abstract syntax tree for a small language
2. Implement context-sensitive, source-level static analyses such as type-checkers or resolving identifiers to identify their binding occurrences.

3. Describe semantic analyses using an attribute grammar.

FPL-Analysis: Program Analysis and Analyzers

Non-Core:

4. Relevant program representations, such as basic blocks, control-flow graphs, def-use chains, and static single assignment.
5. Undecidability and consequences for program analysis
6. Flow-insensitive analysis, such as type-checking and scalable pointer and alias analysis
7. Flow-sensitive analysis, such as forward and backward dataflow analyses
8. Path-sensitive analysis, such as software model checking and software verification
9. Tools and frameworks for implementing analyzers
10. Role of static analysis in program optimization and data dependency analysis during exploitation of concurrency (See also: FPL-Code)
11. Role of program analysis in (partial) verification and bug-finding (See also: FPL-Code)
12. Parallelization
    a. Analysis for auto-parallelization
    b. Analysis for detecting concurrency bugs

Illustrative learning outcomes:
Non-Core:

1. Define useful program analyses in terms of a conceptual framework such as dataflow analysis.
2. Explain the difference between dataflow graph and control flow graph.
3. Explain why non-trivial sound program analyses must be approximate.
4. Argue why an analysis is correct (sound and terminating).
5. Explain why potential aliasing limits sound program analysis and how alias analysis can help.
6. Use the results of a program analysis for program optimization and/or partial program correctness.

FPL-Code: Code Generation

Non-Core:

1. Instruction sets (See also: AR-C)
2. Control flow
3. Memory management (See also: AR-D, OS-F)
4. Procedure calls and method dispatching
5. Separate compilation; linking
6. Instruction selection
7. Instruction scheduling (e.g., pipelining)
8. Register allocation
9. Code optimization as a form of program analysis (See also: FPL-Analysis)
10. Program generation through generative AI,

Illustrative learning outcomes:
Non-Core:

1. Identify all essential steps for automatically converting source code into assembly or other low-level languages.
2. Generate the low-level code for calling functions/methods in modern languages.
3. Discuss why separate compilation requires uniform calling conventions.
4. Discuss why separate compilation limits optimization because of unknown effects of calls.
5. Discuss opportunities for optimization introduced by naive translation and approaches for achieving optimization, such as instruction selection, instruction scheduling, register allocation, and peephole optimization.

FPL-Run-Time: Run-time Behavior and Systems

Non-Core:

1. Process models using stacks and heaps to allocate and deallocate activation records and recovering environment using frame pointers and return addresses during a procedure call including parameter passing examples.
2. Schematics of code lookup using hash tables for methods in implementations of object-oriented programs
3. Data layout for objects and activation records
4. Object allocation in heap
5. Implementing virtual entities and virtual methods; virtual method tables and their application
6. Run-time behavior of object-oriented programs
7. Compare and contrast allocation of memory during information exchange using parameter passing and non-local variables (using chain of static links)
8. Dynamic memory management approaches and techniques: malloc/free, garbage collection (mark-sweep, copying, reference counting), regions (also known as arenas or zones)
9. Just-in-time compilation and dynamic recompilation
10. Interface to operating system (e.g., for program initialization)
11. Interoperability between programming languages including parameter passing mechanisms and data representation (See also: AR-B)
    a. Big Endian, little endian
    b. Data layout of composite data types such as arrays
12. Other common features of virtual machines, such as class loading, threads, and security checking
13. Sandboxing

Illustrative learning outcomes:
Non-Core:

1. Compare the benefits of different memory-management schemes, using concepts such as fragmentation, locality, and memory overhead.
2. Discuss benefits and limitations of automatic memory management.

3. Explain the use of metadata in run-time representations of objects and activation records, such as class pointers, array lengths, return addresses, and frame pointers.
4. Compare and contrast static allocation vs. stack-based allocation vs. heap-based allocation of data elements.
5. Explain why some data elements cannot be automatically deallocated at the end of a procedure/method call (need for garbage collection).
6. Discuss advantages, disadvantages, and difficulties of just-in-time and dynamic recompilation.
7. Discuss use of sandboxing in mobile code
8. Identify the services provided by modern language run-time systems.

FPL-Constructs: Advanced Programming Constructs

Non-Core:

1. Encapsulation mechanisms
2. Lazy evaluation and infinite streams
3. Compare and contrast lazy evaluation vs. eager evaluation
4. Unification vs. assertion vs. expression evaluation
5. Control Abstractions: Exception Handling, Continuations, Monads
6. Object-oriented abstractions: Multiple inheritance, Mixins, Traits, Multimethods
7. Metaprogramming: Macros, Generative programming, Model-based development
8. String manipulation via pattern-matching (regular expressions)
9. Dynamic code evaluation ("eval")
10. Language support for checking assertions, invariants, and pre/post-conditions
11. Domain specific languages, such as database languages, data science languages, embedded computing languages, synchronous languages, hardware interface languages
12. Massive parallel high performance computing models and languages

Illustrative learning outcomes:
Non-Core:

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

FPL-Pragmatics: Language Pragmatics

Non-Core:

1. Effect of technology needs and software requirements on programming language development and evolution
2. Problems domains and programming paradigm
3. Criteria for good programming language design

    a. Principles of language design such as orthogonality
    b. Defining control and iteration constructs
    c. Modularization of large software
4. Evaluation order, precedence, and associativity
5. Eager vs. delayed evaluation
6. Defining control and iteration constructs
7. External calls and system libraries

Illustrative learning outcomes:
Non-Core:

1. Discuss the role of concepts such as orthogonality and well-chosen defaults in language design.
2. Use crisp and objective criteria for evaluating language-design decisions.
3. Implement an example program whose result can differ under different rules for evaluation order, precedence, or associativity.
4. Illustrate uses of delayed evaluation, such as user-defined control abstractions.
5. Discuss the need for allowing calls to external calls and system libraries and the consequences for language implementation.

FPL-Formalism: Formal Semantics

Non-Core:

1. Syntax vs. semantics
2. Approaches to semantics: Axiomatic, Operational, Denotational, Type-based.
3. Axiomatic semantics of abstract constructs such as assignment, selection, iteration using pre-condition, post-conditions and loop invariance
4. Operational semantics analysis of abstract constructs and sequence of such as assignment, expression evaluation, selection, iteration using environment and store
    a. Symbolic execution
    b. Constraint checkers
5. Denotational semantics
    a. Lambda Calculus (See also: AL-Models, FPL-Functional)
6. Proofs by induction over language semantics
7. Formal definitions and proofs for type systems (See also: FPL-Types)
    a. Propositions as types (implication as a function, conjunction as a product, disjunction as a sum)
    b. Dependent types (universal quantification as dependent function, existential quantification as dependent product)
    c. Parametricity

Illustrative learning outcomes:
Non-Core:

1. Construct a formal semantics for a small language.
2. Write a lambda-calculus program and show its evaluation to a normal form.

3. Discuss the different approaches of operational, denotational, and axiomatic semantics.
4. Use induction to prove properties of all programs in a language.
5. Use induction to prove properties of all programs in a language that are well-typed according to a formally defined type system.
6. Use parametricity to establish the behavior of code given only its type.

FPL-Methodologies: Formal Development Methodologies

1. Formal specification languages and methodologies
2. Theorem provers, proof assistants, and logics
3. Constraint checkers See also: FPL-Formalism)
4. Dependent types (universal quantification as dependent function, existential quantification as dependent product) (See also: FPL-Types, FPL-Formalism)
5. Specification and proof discharge for fully verified software systems using pre/post conditions, refinement types, etc.
6. Formal modelling and manual refinement/implementation of software systems
7. Use of symbolic testing and fuzzing in software development
8. Model checking
9. Understanding of situations where formal methods can be effectively applied and how to structure development to maximize their value.

Illustrative learning outcomes:
Non-Core:

1. Use formal modeling techniques to develop and validate architectures.
2. Use proof assisted programming languages to develop fully specified and verified software artifacts.
3. Use verifier and specification support in programming languages to formally validate system properties.
4. Integrate symbolic validation tooling into a programming workflow.
5. Discuss when and how formal methods can be effectively used in the development process.

FPL-Design: Design Principles of Programming Languages

Non-core:

1. Language design principles
    a. simplicity
    b. security
    c. fast translation
    d. efficient object code
    e. orthogonality
    f. readability
    g. completeness
    h. implementation strategies

2. Designing a language to fit a specific domain or problem
3. Interoperability between programming languages,
4. Language portability
5. Formal description of s programming language
6. Green computing principles (See also: SEP-Sustainability)

Illustrative Learning Outcomes:
Non-core:

1. Understand what constitutes good language design and apply that knowledge to evaluate a real programming language.

FPL-Quantum: Quantum Computing

Non-core:

1. Advantages and disadvantages of quantum computing
2. Qubit and qubit state
3. superposition and interference
4. entanglement
5. Quantum algorithms (e.g., Shor’s, Grover’s algorithms)

Illustrative Learning Outcomes:
Non-core

1. An appreciation of quantum computing and its application to certain problems.
2. An appreciation of classical computing and its role as quantum computing emerges.

FPL-SEP: Society, Ethics and Professionalism

Non-Core:

1. Impact of English-centric programming languages
2. Enhancing accessibility and inclusivity for people with disabilities
    a. Supporting assistive technologies
3. Human factors related to programming languages and usability
    a. Impact of syntax on accessibility
    b. Supporting cultural differences (e.g., currency, decimals, dates)
    c. Neurodiversity
4. Etymology of terms such as “class”, “master”, “slave” in programming languages
5. Increasing accessibility by supporting multiple languages within applications (UTF)

Illustrative learning outcomes:
Non-Core:

1. Consciously design programming languages to be inclusive and non-offensive

Professional Dispositions

1. Meticulous: Students must demonstrate and apply the highest standards when using programming languages and formal methods to build safe systems that are fit for their purpose.
2. Meticulous: Attention to detail is essential when using programming languages and applying formal methods.
3. Inventive: Programming and approaches to formal proofs is inherently a creative process, students must demonstrate innovative approaches to problem solving. Students are accountable for their choices regarding the way a problem is solved.
4. Proactive: Programmers are responsible for anticipating all forms of user input and system behavior and to design solutions that address each one.
5. Persistent: Students must demonstrate perseverance since the correct approach is not always self-evident and a process of refinement may be necessary to reach the solution.

Math Requirements

Required:

- Discrete Mathematics – Boolean algebra, proof techniques, digital logic, sets and set operations, mapping, functions and relations, states and invariants, graphs and relations, trees, counting, recurrence relations, finite state machine, regular grammar
- Logic – propositional logic (negations, conjunctions, disjunctions, conditionals, biconditionals), first-order logic, logical reasoning (induction, deduction, abduction).
- Mathematics – complex numbers, matrices, linear transformation, probability, statistics

Course Packaging Suggestions

Programming Language Concepts (Introduction) Course to include the following:

● FPL-OOP: Object-Oriented Programming	5 CS Core hours 1 KA Core hours
● FPL-Functional: Functional Programming	4 CS Core hours 3 KA Core hours
● FPL-Logic: Logic Programming	3 KA Core hours
● FPL-Compiled-Interpret: Compiled vs Interpreted Programming	1 CS Core hour
● FPL-Scripting: Scripting	2 CS Core hours
● FPL-Event-Driven: Event-Driven and Reactive Programming	2 CS Core hours 2 KA Core hours
● FPL-Parallel: Parallel and Distributed Computing	3 CS Core hours 2 KA Core hours
● FPL-Types: Type Systems	3 CS Core hours 4 KA Core hours
● FPL-Systems: Systems Execution and Memory Model	

	3 CS Core hours
FPL-Translation: Language Translation and Execution	2 KA Core hours
FPL-Abstraction: Program Abstraction and Representation	3 KA Core hours
FPL-Quantum: Quantum Computing	2 CS Core hours
FPL-SEP: FPL and SEP	1 Non-Core hour

Pre-requisites:

- Discrete Mathematics – Boolean algebra, proof techniques, digital logic, sets and set operations, mapping, functions and relations, states and invariants, graphs and relations, trees, counting, recurrence relations, finite state machine, regular grammar.

Programming Language Implementation (Advanced) Course to include the following:

FPL-Types: Type Systems	3 Non-core hours
FPL-Translation: Language Translation and Execution	3 Non-core hours
FPL-Syntax: Syntax Analysis	3 Non-core hours
FPL-Semantics: Compiler Semantic Analysis	5 Non-core hours
FPL-Analysis: Program Analysis and Analyzers	5 Non-core hours
FPL-Code: Code Generation	5 Non-core hours
FPL-Run-Time: Run-time Systems	4 Non-core hours
FPL-Constructs: Advanced Programming Constructs	4 Non-core hours
FPL-Pragmatics: Language Pragmatics	3 Non-core hours
FPL-Formalism: Formal Semantics	5 Non-core hours
FPL-Methodologies: Formal Development Methodologies	5 Non-core hours

Pre-requisites:

- Discrete mathematics – Boolean algebra, proof techniques, digital logic, sets and set operations, mapping, functions and relations, states and invariants, graphs and relations, trees, counting, recurrence relations, finite state machine, regular grammar.
- Logic – propositional logic (negations, conjunctions, disjunctions, conditionals, biconditionals), first-order logic, logical reasoning (induction, deduction, abduction).
- Introductory programming course.
- Programming proficiency in programming concepts such as:
  - type declarations such as basic data types, records, indexed data elements such as arrays and vectors, and class/subclass declarations, types of variables,
  - scope rules of variables,
  - selection and iteration concepts, function and procedure calls, methods, object creation
- Data structure concepts such as:
  - abstract data types, sequence and string, stack, queues, trees, dictionaries
  - pointer-based data structures such as linked lists, trees and shared memory locations
  - Hashing and hash tables
- System fundamentals and computer architecture concepts such as:
  - Digital circuits design, clocks, bus

    - registers, cache, RAM and secondary memory
    - CPU and GPU
- Basic knowledge of operating system concepts such as:
    - Interrupts, threads and interrupt-based/thread-based programming
    - Scheduling, including prioritization
    - Memory fragmentation
    - Latency

Committee

Chair: Michael Oudshoorn, High Point University, High Point, NC, USA
Members:

- Annette Bieniusa, TU Kaiserslautern, Kaiserslautern, Germany
- Brijesh Dongol, University of Surrey, Guildford, UK
- Michelle Kuttel, University of Cape Town, Cape Town, South Africa
- Doug Lea, State University of New York at Oswego, Oswego, NY, USA
- James Noble, Victoria University of Wellington, Wellington, New Zealand
- Mark Marron, Microsoft Research, Seattle, WA, USA
- Peter-Michael Osera, Grinnell College, Grinnell, IA, USA
- Michelle Mills Strout, University of Arizona, Tucson, AZ, USA

Contributors:

- Alan Dearle, University of St. Andrews, St. Andrews, Scotland

Graphics and Interactive Techniques (GIT)

Preamble

Computer graphics is the term used to describe the computer generation and manipulation of images and can be viewed as the science of enabling visual communication through computation. Its applications include machine learning; medical imaging; engineering; scientific, information, and knowledge visualization; cartoons; special effects; simulators; and video games. Traditionally, graphics at the undergraduate level focused on rendering, linear algebra, physics, the graphics pipeline, interaction, and phenomenological approaches. Today's graphics courses increasingly include physical computing, animation, and haptics. At the advanced level, undergraduate institutions are increasingly likely to offer one or more courses specializing in a specific graphics knowledge unit: e.g., gaming, animation, visualization, tangible or physical computing, and immersive courses such as AR/VR/XR. There is considerable overlap with other computer science knowledge areas: Algorithmic Foundations, Architecture and Organization, Artificial Intelligence; Human-Computer Interaction; Parallel and Distributed Computing; Specialized Platform Development; Software Engineering, and Society, Ethics, and Professionalism.

In order for students to become adept at the use and generation of computer graphics, many issues must be addressed, such as human perception and cognition, data and image file formats, hardware interfaces, and application program interfaces (APIs). Unlike other knowledge areas, knowledge units or topics within Graphics and Interactive Techniques may be included in a variety of elective courses ranging from a traditional Interactive Computer Graphics course to Gaming or Animation. Alternatively, graphics topics may be introduced in an applied project in Human Computer Interaction, Embedded Systems, Web Development, etc. Undergraduate computer science students who study the knowledge units specified below through a balance of theory and applied instruction will be able to understand, evaluate, and/or implement the related graphics and interactive techniques as users and developers. Because technology changes rapidly, the Graphics and Interactive Techniques subcommittee attempted to avoid being overly prescriptive. Where provided, examples of APIs, programs, and languages should be considered as appropriate examples in 2023. In effect, this is a snapshot in time.

Graphics as a knowledge area has expanded and become pervasive since the CS2013 report. Machine learning, computer vision, data science, artificial intelligence, and interfaces driven by embedded sensors in everything from cars to coffee makers use graphics and interactive techniques. The now ubiquitous smartphone has made the majority of the world's population regular users and creators of graphics, digital images, and the interactive techniques to manipulate them. Animations, games, visualizations, and immersive applications that ran on desktops in 2013, now can run on mobile devices. The amount of stored digital data grew exponentially since 2013, and both data and visualizations are now published by myriad sources including news media and scientific organizations.

Revenue from mobile video games now exceeds that of music and movies combined.[1] Computer Generated Imagery (CGI) is employed in almost all films.

It is critical that students and faculty confront the ethical questions and conundrums that have arisen and will continue to arise because of applications in computer graphics—especially those that employ machine learning, data science, and artificial intelligence. Today's news unfortunately provides examples of inequity and wrong-doing in autonomous navigation, deepfakes, computational photography, and facial recognition.

Changes since CS 2013

In an effort to align CC2013's Graphics and Visualizations areas with SIGGRAPH, we have renamed it Graphics and Interactive Techniques (GIT). To capture the expanding footprint of the field, the following knowledge units have been added to the original list of knowledge units: Fundamental Concepts, Visualization, Basic Rendering (renamed Rendering), Geometric Modeling, Advanced Rendering (renamed Shading), and Computer Animation:

- Immersion (MR, AR, VR)
- Interaction
- Image Processing
- Tangible/Physical Computing
- Simulation

Core Hours

Knowledge Unit	CS Core	KA Core
Fundamental Concepts	4	
Rendering		18
Geometric Modeling		6
Shading		6
Computer Animation		Animation KA - 6
Visualization		Visualization KA - 6
Immersion (MR, AR, VR)		Immersion KA - 6
Interaction		Interaction KA - 6

[1] Jon Quast, Clay Bruning, and Sanmeet Deo. "Markets: This Opportunity for Investors Is Bigger Than Movies and Music Combined." retrieved from https://www.nasdaq.com/articles/this-opportunity-for-investors-is-bigger-than-movies-and-music-combined-2021-10-03.

Image Processing		Image Processing KA - 6
Tangible/Physical Computing		Tangible/Physical Computing KA - 6
Simulation		Simulation KA - 6
Total	**4**	

Knowledge Units

GIT-Fundamentals: Fundamental Concepts

For nearly every computer scientist and software developer, 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. Note that many of these topics are revisited in greater depth in later sections.

CS Core:

1. Uses of graphics and interactive techniques and potential risks and abuses
    a. Entertainment, business, and scientific applications: examples include visual effects, machine learning, computer vision, user interfaces, video editing, games and game engines, computer-aided design and manufacturing, data visualization, and virtual/augmented/mixed reality.
    b. Intellectual property, deep fakes, facial recognition, privacy (See also: SEP-Privacy, SEP-IP, SEP-Ethics)
2. Graphic output
    a. displays (e.g., LCD)
    b. printer
    c. analog film
    d. resolution
        i. pixels for visual displays
        ii. dots for laser printers
    e. aspect ratio
    f. frame rate
3. Human vision system
    a. tristimulus reception (RGB)
    b. eye as a camera (projection)
    c. persistence of vision (frame rate, motion blur)
    d. contrast (detection, Mach banding, dithering/aliasing)
    e. non-linear response (dynamic range, tone mapping)
    f. binocular vision (stereo)
    g. accessibility (color deficiency, strobing, monocular vision, etc.) (See also: SEP-IDEA, HCI-User)
4. Standard image formats
    a. raster

        i. lossless (e.g., TIF)
        ii. lossy (e.g., JPG, PNG, etc.)
    b. vector (e.g. SVG, Adobe Illustrator)
5. Digitization of analog data
    a. rasterization
    b. resolution
    c. sampling and quantization
6. Color models: additive (RGB), subtractive (CMYK), and color perception (HSV)
7. Tradeoffs between storing image data and re-computing image data.
8. Applied interactive graphics, e.g., graphics API, mobile app
9. Animation as a sequence of still images

Illustrative Learning Outcomes:

CS Core:

1. Identify common uses of digital presentation to humans (e.g., computer graphics, sound).
2. Explain how analog signals can be reasonably represented by discrete samples, for example, how images can be represented by pixels.
3. Compute the memory requirement for storing a color image given its resolution.
4. Create a graphic depicting how the limits of human perception affect choices about the digital representation of analog signals.
5. Design a user interface and an alternative for persons with color-perception deficiency.
6. Construct a simple graphical user interface using a standard API.
7. When should you use each of the following common graphics file formats: JPG, PNG, MP3, MP4, and GIF? Why?
8. Give an example of a lossy- and a lossless-image compression technique found in common graphics file formats.
9. Describe color models and their use in graphics display devices.
10. Compute the memory requirements for a multi-second movie (lasting *n* seconds) displaying at a specific framerate (*f* frames per second) at a specified resolutions (*r* pixels per frame)
11. Compare and contrast digital video to analog video.
12. Describe the basic process of producing continuous motion from a sequence of discrete frames (sometimes called "flicker fusion").
13. Describe a possible visual misrepresentation that could result from digitally sampling an analog world.
14. Compute memory space requirements based on resolution and color coding.
15. Compute time requirements based on refresh rates and rasterization techniques.

GIT-Visualization

KA Core

1. Data Visualization and Information Visualization
2. Visualization of:
    a. 2D/3D scalar fields

    b. Vector fields and flow data
    c. Time-varying data
    d. High-dimensional data
    e. Non-spatial data
2. Visualization techniques (color mapping, isosurfaces, dimension reduction, parallel coordinates, multi-variate, tree/graph-structured, text)
3. Direct volume data rendering: ray-casting, transfer functions, segmentation.
4. Common data formats (e.g., HDF, netCDF, geotiff, raw binary, CSV, ASCII to parse, etc.)
5. Common Visualization software and libraries (e.g., R, Processing, D3.js, GIS, Matlab, IDL, Python, etc.)
6. Perceptual and cognitive foundations that drive visual abstractions
    a. Visual communication
    b. Color theory
    7. Visualization design
    a. Purpose (discovery, outreach)
    b. Audience (technical, general public)
    c. Ethically responsible visualization
        i. Avoid misleading visualizations (due to exaggeration, hole filling, smoothing, data cleanup).
        ii. Even correct data can be misleading, e.g., aliasing, incorrectly moving or stopped fan blades.
8. Evaluation of visualization methods and applications
9. Visualization bias
10. Applications of visualization

Illustrative Learning Outcomes

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

GIT-Rendering

This section describes basic rendering and fundamental graphics techniques that nearly every undergraduate course in graphics will cover and that are essential for further study in most graphics-related courses.

KA Core (See SPD-Game 2 & 3c)

1. Object and scene modeling

    a. Object representations: polygonal, parametric, etc.
    b. Modeling transformations: affine and coordinate-system transformations
    c. Scene representations: scene graphs
2. Camera and projection modeling
    a. Pinhole cameras, similar triangles, and projection model
    b. Camera models
    c. Projective geometry
3. Radiometry and light models
    a. Radiometry
    b. Rendering equation
    c. Rendering in nature – emission and scattering, etc.
4. Rendering
    a. Simple triangle rasterization
    b. Rendering with a shader-based API
    c. Visibility and occlusion, including solutions to this problem (e.g., depth buffering, Painter's algorithm, and ray tracing)
    d. Texture mapping, including minification and magnification (e.g., trilinear MIP mapping)
    e. Application of spatial data structures to rendering.
    f. Ray tracing
    g. Sampling and anti-aliasing

Illustrative Learning Outcomes

1. Diagram 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).
2. Describe the basic rendering pipeline.
3. Compare and contrast how forward and backwards rendering factor into the graphics pipeline.
4. Create a program to display 2D shapes in a window.
5. Create a program to display 3D models.
6. Derive linear perspective from similar triangles by converting points (x, y, z) to points (x/z, y/z, 1).
7. Compute two-dimensional and 3-dimensional points by applying affine transformations.
8. Apply the three-dimensional coordinate system and the changes required to extend 2D transformation operations to handle transformations in 3D.
9. Explain the concept and applications of texture mapping, sampling, and anti-aliasing.
10. Compare ray tracing and rasterization for the visibility problem.
11. Implement simple procedures that perform transformation and clipping operations on simple two-dimensional shapes.
12. Implement a simple real-time renderer using a rasterization API (e.g., OpenGL, webGL) using vertex buffers and shaders.
13. Compare and contrast the different rendering techniques.
14. Compare and contrast the difference in transforming the camera vs. the models.

GIT-Modeling: Geometric Modeling

KA Core

1. Basic geometric operations such as intersection calculation and proximity tests on 2D objects
2. Surface representation/model
    a. Tessellation
    b. Mesh representation, mesh fairing, and mesh generation techniques such as Delaunay triangulation, and marching cubes/tetrahedrons
    c. Parametric polynomial curves and surfaces
    d. Implicit representation of curves and surfaces
    e. Spatial subdivision techniques
3. Volumetric representation/model
    a. Volumes, voxels, and point-based representations.
    b. Signed Distance Fields
    c. Sparse Volumes, i.e., VDB
    d. Constructive Solid Geometry (CSG) representation
4. Procedural representation/model (See also: FPL-Translation 1.a)
    a. Fractals
    b. L-Systems
5. Multi-resolution modeling (See also SPD-Game)
6. Reconstruction, e.g., 3D scanning, photogrammetry, etc.

Illustrative Learning Outcomes:

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

GIT-Shading

Topics:

1. Solutions and approximations to the rendering equation, for example:
    a. Distribution ray tracing and path tracing
    b. Photon mapping
    c. Bidirectional path tracing
    d. Metropolis light transport
2. Time (motion blur), lens position (focus), and continuous frequency (color) and their impact on rendering
3. Shadow mapping
4. Occlusion culling
5. Bidirectional Scattering Distribution function (BSDF) theory and microfacets
6. Subsurface scattering
7. Area light sources

8. Hierarchical depth buffering
9. Image-based rendering
10. Non-photorealistic rendering
11. GPU architecture (See also AR-Heterogeneous Architectures 1 & 3)
12. Human visual systems including adaptation to light, sensitivity to noise, and flicker fusion (See also: HCI-Accessibility, SEP-IDEA)

Illustrative 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 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. Show 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.

GIT-Animation: Computer Animation

KU Core:

1. Principles of Animation: Squash and Stretch, Timing, Anticipation, Staging, Follow Through and Overlapping Action, Straight Ahead Action and Pose-to-Pose Action, Slow In and Out, Arcs, Exaggeration, and Appeal
2. Types of animation
   a. 2- and 3-dimensional animation
   b. motion graphics
   c. motion capture
   d. motion graphics
   e. stop animation
3. Key-frame animation
   a. Keyframe Interpolation Methods: Lerp / Slerp / Spline
4. Forward and inverse kinematics (See also: SPD-Robot)
5. Skinning algorithms
   a. Capturing
   b. Linear blend, dual quaternion
   c. Rigging
   d. Blend shapes
   e. Pose space deformation
6. Motion capture
   a. Set up and fundamentals
   b. Blending motion capture clips

c. Blending motion capture and keyframe animation
d. Ethical considerations (e.g., accessibility and privacy)
    i. Avoidance of “default” captures - there is no typical human walk cycle.

Illustrative Learning Outcomes:

1. Using a simple open-source character model and rig, illustrate why each of the principles of animation is fundamental to realistic animation.
2. Compute the location and orientation of model parts using a forward kinematic approach.
3. Compute the orientation of articulated parts of a model from a location and orientation using an inverse kinematic approach.
4. Compare the tradeoffs in different representations of rotations.
5. Implement the spline interpolation method for producing in-between positions and orientations.
6. Use off-the-shelf animation software to construct, rig, and animate simple organic forms.

GIT: Simulation

Simulation has strong ties to Computational Science. In the graphic domain, however, simulation techniques are re-purposed to a different end. Rather than creating predictive models, the goal instead is to achieve a mixture of physical plausibility and artistic intention. To illustrate, the goals of “model surface tension in a liquid” and “produce a crown splash” are related, but different. Depending on the simulation goals, covered topics may vary as shown below.

Goals (Given a goal, which topics should be used):

- Particle systems
    - Integration methods (Forward Euler, Midpoint, Leapfrog)
- Rigid Body Dynamics
    - Particle systems
    - Collision Detection
        - Tri/point, edge/edge
- Cloth
    - Particle systems
    - Mass/spring networks
    - Collision Detection
        - Tri/point, edge/edge
- Particle-Based Water
    - Integration methods
    - Smoother Particle Hydrodynamics (SPH) Kernels
    - Signed Distance Function-Based Collisions
- Grid-Based Smoke and Fire
    - Semi-Lagrangian Advection
    - Pressure Projection
- Grid and Particle-Based Water
    - Particle-Based Water

- Grid-Based Smoke and Fire
    - Semi-Lagrangian Advection
    - Pressure Projection
- Grid and Particle-Based Water
    - Particle-Based Water
    - Grid-Based Smoke, and Fire

KU Core:

1. Collision detection and response
    a. Signed Distance Fields
    b. Sphere/sphere
    c. Triangle/point
    d. Edge/edge
2. Procedural animation using noise
3. Particle systems
    a. Integration methods (Forward Euler, Midpoint, Leapfrog)
    b. Mass/spring networks
    c. Position-based dynamics
    d. Rules (boids/crowds)
    e. Rigid bodies
4. Grid-based fluids
    a. Semi-Lagrangian advection
    b. Pressure Projection

- Heightfields
    a. Terrain: Transport, erosion
    b. Water: Ripple, Shallow water.
- Rule-based systems, e.g., L-Systems, Space-colonizing systems, Game of Life, etc.

Illustrative Learning Outcomes:

1. 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.
2. Contrast the basic ideas behind fluid simulation methods for modeling ballistic trajectories, for example for splashes, dust, fire, or smoke.
3. Implement a smoke solver with user interaction

GIT-Immersion

KU Core: See also: SPD-Games, SPD-Mobile, HCI-Design

1. Define and distinguish VR, AR, and MR
2. Define and distinguish immersion and presence
3. 360 Video
4. Stereoscopic display
    a. Head-mounted displays

    b. Stereo glasses
5. Viewer tracking
    a. Inside out vs Outside In
    b. Head / Body / Hand / tracking
6. Time-critical rendering to achieve optimal motion to photon (MTP) latency
    a. Multiple levels of details (LOD)
    b. Image-based VR
    c. Branching movies
7. Distributed VR, collaboration over computer network
8. Applications in medicine, simulation, training, and visualization
9. Safety in immersive applications
    a. Motion sickness
    b. VR obscures the real world, which increases the potential for falls and physical accidents

Illustrative Learning Outcomes:

1. Create a stereoscopic image.
2. Summarize the pros and cons of different types of viewer tracking.
3. Compare and contrast the differences between geometry- and image-based virtual reality.
4. Judge and defend the design issues of user action synchronization and data consistency in a networked environment.
5. Create the specifications for an augmented reality application to be used by surgeons in the operating room.
6. Evaluate an immersive application's accessibility (cross-reference with HCI)
7. Identify the most important technical characteristics of a VR system/application that should be controlled to avoid motion sickness and explain why.

GIT-Interaction

Interactive computer graphics is a requisite part of real-time applications ranging from the utilitarian-like word processors to virtual and/or augmented reality applications. Students will learn the following topics in a graphics course or a course that covers HCI/GUI Construction and HCI/Programming.

KU Core:

1. Event Driven Programming (See also: FPL-Event-Driven)
    a. Mouse or touch events
    b. Keyboard events
    c. Voice input
    d. Sensors
    e. Message passing communication
    f. Network events
2. Graphical User Interface (Single Channel)
    a. Window
    b. Icons
    c. Menus

    d. Pointing Devices
3. Gestural Interfaces (See also: SPD-Games)
    a. Touch screens
    b. Game controllers
4. Haptic Interfaces
    a. External actuators
    b. Gloves
    c. Exoskeletons
5. Multimodal Interfaces
6. Head-worn Interfaces (AI)
    a. brainwave (EEG type electrodes)
    b. headsets with embedded eye tracking
    c. AR glasses
7. Natural Language Interfaces (See also: AI-NLP)
8. Accessibility (See also: SEP_IDEA)

Illustrative Learning Outcomes:

1. Create a simple game that responds to single channel mouse and keyboard events
2. Program a circuit to respond to a variable resistor
3. Create a mobile app that responds to touch events
4. Use gestures to control a program
5. Design and implement an application that provides haptic feedback
6. Design and implement an application that responds to different event triggers

GIT-Image: Image Processing

Image Processing consists of the analysis and processing of images for multiple purposes, but most frequently to improve image quality and to manipulate imagery. It is the cornerstone of Computer Vision which is a KU in AI.

KU Core:

1. Morphological operations
    a. Connected components
    b. Dilation
    c. Erosion
    d. Computing region properties (area, perimeter, centroid, etc.)
2. Color histograms
    a. Representation
    b. Contrast enhancement through normalization
3. Image enhancement
    a. Convolution
    b. Blur (e.g., Gaussian)
    c. Sharpen (e.g.,Laplacian)
    d. Frequency filtering (e.g.,low-pass, high-pass)
4. Image restoration

    a. Noise, degradation
    b. Inpainting and other completion algorithms
    c. Wiener filter
5. Image coding
    a. Redundancy
    b. Compression (e.g., Huffman coding)
    c. DCT, wavelet transform, Fourier transforms
    d. Nyquist Theorem
    e. Watermarks
6. Connections to deep learning (e.g., Convolutional Neural Networks) (See AI-ML 7)

Illustrative Learning Outcomes:
1. Use dilation and erosion to smooth the edges of a binary image.
2. Manipulate the hue of an image
3. Filter an image using a high-pass filter (advanced: in frequency domain)
4. Restore missing parts of an image using an in-paint algorithm (e.g., Poisson image editing)
5. Enhance an image by selectively filtering in the frequency domain

GIT-Physical: Tangible/Physical Computing

Cross-cutting with Specialized Platform Development and HCI
KU Core:
1. Interaction with the physical world
    a. Acquisition of data from sensors
    b. Driving external actuators
2. Connection to physical artifacts
    a. Computer-Aided Design (CAD)
    b. Computer-Aided Manufacturing (CAM)
    c. Fabrication (See also HCI-Design)
        i. HCI prototyping (see HCI)
        ii. Additive (3D printing)
        iii. Subtractive (CNC milling)
        iv. Forming (vacuum forming)
3. Internet of Things (See also SDP-Interactive)
    a. Network connectivity
    b. Wireless communication

Illustrative Learning Outcomes:
1. Construct a simple switch and use it to turn on an LED.
2. Construct a simple system to move a servo in response to sensor data
3. Use a light sensor to vary a property of something else (e.g. color or brightness of an LED or graphic)
4. Create a 3D form in a CAD package
    a. Show how affine transformations are achieved in the CAD program

    b. Show an example of instances of an object
    c. Create a fabrication plan. Provide a cost estimate for materials and time. How will you fabricate it?
    d. Fabricate it. How closely did your actual fabrication process match your plan? Where did it differ?
5. Write the G- and M-Code to construct a 3D maze, and use a CAD/CAM package to check your work
6. If you were to design an IoT pill dispenser, would you use Ethernet, WiFi, Bluetooth, RFID/NFC, or something else for Internet connectivity. Why? Make one.
7. Distinguish between the different types of fabrication and describe when you would use each.

GIT-SEP: Society, Ethics and Professionalism

KU Core:

1. Physical Computing
    a. Privacy (e.g., health and other personal information)
2. Accessibility in immersive applications (See also: SEP-IDEA)
    a. Accessible to those who cannot move
    b. Accessible to those who cannot be moved
3. Ethics/privacy in immersive applications. (See also: SEP-Privacy, SEP-Ethics, and SEP-Security)
    a. Acquisition of private data (room scans, body proportions, active cameras, etc)
    b. Can’t look away from immersive applications easily.
    c. Danger to self/surroundings while immersed
4. Bias in image processing
    a. Deep fakes
    b. Applications that misidentify people based on skin color or hairstyle
5. Copyright law and watermarked images

Illustrative Learning Outcomes:

1. Discuss the security issues inherent in location tags.
2. Describe the ethical pitfalls of facial recognition. Can facial recognition be used ethically? If so, how?
3. Discuss the copyright issues of using watermarked images to train a neural network.

Professional Dispositions

1. Self-directed: self-learner, self-motivated. It is important for graphics programmers to keep up with technical advances.
2. Collaborative: team player: Graphics programmers typically develop on teams composed of people with differing specialties.
3. Effective communication is critical.
    a. oral
    b. written

    c. code

Math Requirements

Required:

1. Linear Algebra:
    a. Points (coordinate systems & homogeneous coordinates), vectors, and matrices
    b. Vector operations: addition, scaling, dot and cross products
    c. Matrix operations: addition, multiplication, determinants
    d. Affine transformations
2. Calculus
    a. Continuity

Desirable:

1. Linear Algebra
    a. Eigenvectors and Eigen decomposition
    b. Gaussian Elimination and Lower Upper Factorization
    c. Singular Value Decomposition
2. Calculus
    a. Quaternions
3. Probability

Necessary and Desirable Data Structures:

1. Data Structures necessary for this knowledge area include:
    a. Directed Acyclic Graphs
    b. Tuples (Points / vectors / matrices of fixed dimension)
    c. Dense 1d, 2d, 3d arrays.
2. Data Structures desirable for this knowledge area include:
    a. Array of Structures vs. Structure of Arrays
    b. Trees (e.g., K-trees, quadtrees, Huffman Trees)

Shared Topics and Crosscutting Themes

Shared Topics:

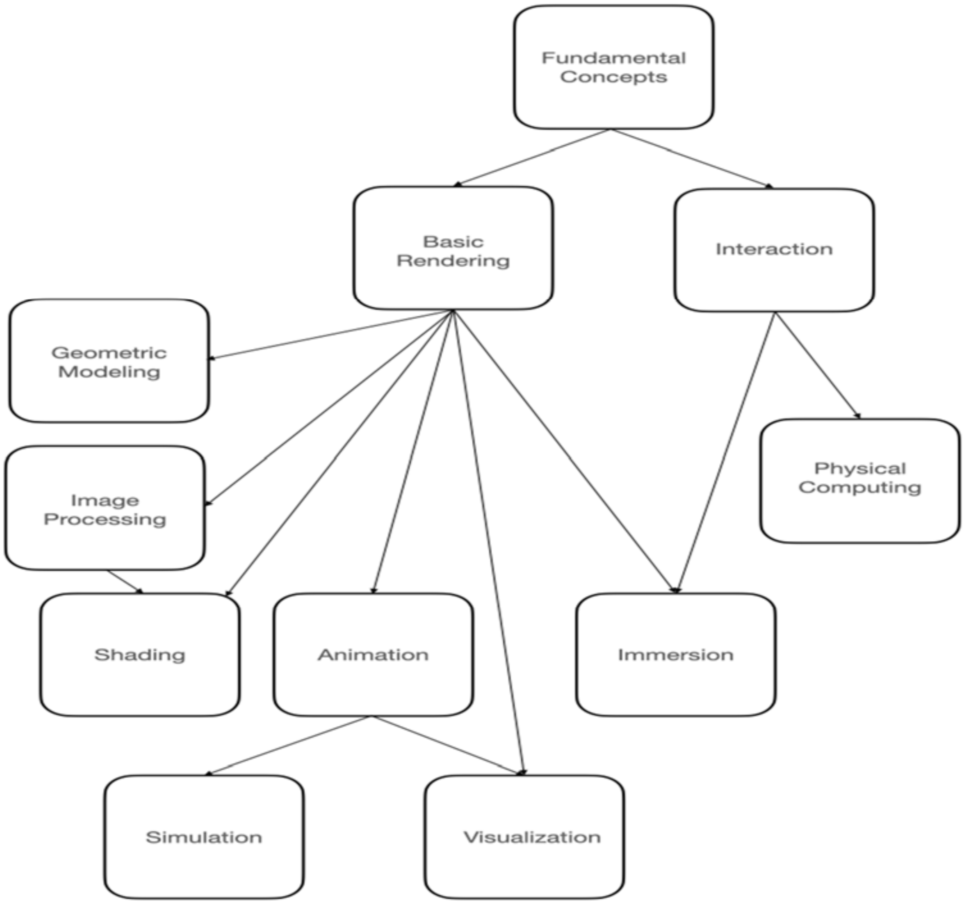
- GIT-Immersion □ □ HCI
- GIT-Interaction □ □ ~~HCI/GUI Programming~~
- Graftals: GIT-Geometric Modeling □ □ FPL-H: Language Translation and Execution
- Simulation
- Visualization in GIT, AI, and Specialized Platform Development Interactive Computing Platforms (Data Visualization)
- Image Processing in GIT and Specialized Platform Development Interactive Computing Platforms (Supporting Math Studies)
- Tangible Computing in GIT and Specialized Platform Development Interactive Computing Platforms (Game Platforms)

- Tangible Computing in GIT and Specialized Platform Development Interactive Computing Platforms (Embedded Platforms)
- Tangible Computing and Animation in GIT and Specialized Platform Development Interactive Computing Platforms (Robot Platforms)
- Immersion in GIT and Specialized Platform Development Interactive Computing Platforms (Mobile Platforms)
- Image Processing in GIT and Advanced Machine Learning (Graphical Models) in AI
- Image Processing and Physical Computing in GIT and Robotics Location and Mapping and Navigation in AI
- Image Processing in GIT and Perception and Computer Vision in AI
- Core Graphics in GIT and Algorithms and Application Domains in PD
- GIT and Interactive Computing Platforms in SPD
- GIT and Game Platforms in SPD
- GIT and Imbedded Platforms in SPD

Crosscutting themes:

- Efficiency
- Ethics
- Modeling
- Programming
- Prototyping
- Usability
- Evaluation

Course Packaging Suggestions



Interactive Computer Graphics to include the following:

- GIT-Rendering: 40 hours
- SEP-C: Professional Ethics: 4 hours

Pre-requisites:

- CS2
- Affine Transforms from Linear Algebra
- Trigonometry

Skill statement: A student who completes this course should understand and be able to create basic computer graphics using an API. They should know how to position and orient models, the camera, and distant and local lights.

Tangible Computing to include the following:

- GIT-Tangible/Physical Computing (27 hours)
- SPD-E: Embedded Platforms (10 hours)
- HCI-User: Understanding the User (3 hours)
- SEP-Privacy: Privacy and Civil Liberties and SEP-K: Diversity, Equity and Inclusion (4 hours)

Pre-requisites:

- CS1

Skill statement: A student who completes this course should be able to design and build circuits and program a microcontroller. They will understand polarity, Ohm's law, and how to work with electronics safely.

Image Processing to include the following:

- GIT-Image Processing (30 hours)
- GIT-Rendering: Basic Rendering: (10 hours)
- SEP-Privacy: Privacy and Civil Liberties,SEP-IDEA: Inclusion, Diversity, Equity and Accessibility and SEP-IP: Intellectual Property (4 hours)

Pre-requisites:

- CS2
- Linear Algebra
- Probability

Skill statement: A student who completes this course should understand and be able to appropriately acquire, process, display, and save digital images.

Data Visualization to include the following:

- GIT-Visualization (30 hours)
- GIT-Rendering: Basic Rendering (10 hours)
- HCI-User: Understanding the User (3 hours)
- SEP-E: Privacy and Civil Liberties and SEP-IDEA: Inclusion, Diversity, Equity and Accessibility, SEP-Ethics: Professional Ethics (4 hours)

Pre-requisites:

- CS2
- Linear Algebra

Skill statement: A student who completes this course should understand how to select a dataset; ensure the data are accurate and appropriate; design, develop and test a visualization program that depicts the data and is usable.

Simulation to include the following:

- GIT-Simulation (30 hours)
- GIT-Rendering (10 hours)
- SEP-Ethics: Professional Ethics (4 hours)

Pre-requisites:

- CS2
- Linear Algebra

Skill statement: A student who completes this course should understand and be able to create directable simulations, both of physical and non-physical systems.

Committee

Chair: Susan Reiser, UNC Asheville, Asheville, USA

Members:

- Erik Brunvand, University of Utah, Salt Lake City, USA
- Kel Elkins, NASA/GSFC Scientific Visualization Studio, Greenbelt, MD
- Jeff Lait, SideFX, Toronto, Canada
- Amruth Kumar, Ramapo College, Mahwah, USA
- Paul Mihail, Valdosta State University, Valdosta, USA
- Tabitha Peck, Davidson College, Davidson, USA
- Ken Schmidt, NOAA NCEI, Asheville, USA
- Dave Shreiner, UnityTechnologies & Sonoma State University, San Francisco, USA

Contributors:

- Ginger Alford, Southern Methodist University, TX, USA
- Christopher Andrews, Middlebury College, Middlebury, VT, USA
- AJ Christensen, NASA/GSFC Scientific Visualization Studio – SSAI, Champaign, IL
- Roger Eastman, University of Maryland, College Park, MD, USA
- Ted Kim, Yale University, New Haven, CT, USA
- Barbara Mones, University of Washington, Seattle, WA, USA
- Greg Shirah, NASA/GSFC Scientific Visualization Studio, Greenbelt, MD
- Beatriz Sousa Santos, University of Aveiro, Portugal
- Anthony Steed, University College, London, UK

Human-Computer Interaction (HCI)

Preamble

Computational systems not only enable users to solve problems, but also foster social connectedness and support a broad variety of human endeavors. Thus, these systems should interact with their users and solve problems in ways that respect individual dignity, social justice, and human values and creativity. Human-computer interaction (HCI) addresses those issues from an interdisciplinary perspective that includes psychology, business strategy, and design principles.

Each user is different and, from the perspective of HCI, the design of every system that interacts with people should anticipate and respect that diversity. This includes not only accessibility, but also cultural and societal norms, neural diversity, modality, and the responses the system elicits in its users. An effective computational system should evoke trust while it treats its users fairly, respects their privacy, provides security, and abides by ethical principles.

These goals require design-centric engineering that begins with intention and with the understanding that design is an iterative process, one that requires repeated evaluation of its usability and its impact on its users. Moreover, technology evokes user responses, not only by its output, but also by the modalities with which it senses and communicates. This knowledge area heightens the awareness of these issues and should influence every computer scientist.

Changes since CS 2013

Driven by this broadened perspective, the HCI knowledge area has revised the CS 2013 document in several ways:

- Knowledge units have been renamed and reformulated to reflect current practice and to anticipate future technological development.
- There is increased emphasis on the nature of diversity and the centrality of design focused on the user.
- Modality (e.g., text, speech) is still emphasized given its key role throughout HCI, but with a reduced emphasis on particular modalities in favor of a more timely and empathetic approach.
- The curriculum reflects the importance of understanding and evaluating the impacts and implications of a computational system on its users, including issues in ethics, fairness, trust, and explainability.
- Given its extensive interconnections with other knowledge areas, we believe HCI is itself a cross-cutting knowledge area with connections to Artificial Intelligence; Society, Ethics and Professionalism, Software Development Fundamentals, Software Engineering.

Core Hours

Knowledge Unit	CS Core	KA Core
Understanding the User	2	5

Accountability and Responsibility in Design	2	2
Accessibility and Inclusive Design	2	2
Evaluating the Design	1	2
System Design	1	5
Total Hours	**8**	**16**

Knowledge Units

HCI-User: Understanding the User: Individual goals and interactions with others

CS Core:

1. User-centered design and evaluation methods:
    a. "you are not the users"
    b. user needs-finding
    c. formative studies
    d. interviews
    e. surveys
    f. usability tests

KA Core:

2. User-centered design methodology: (See also: SE-Tools)
    a. personas/persona spectrum
    b. user stories/storytelling and techniques for gathering stories
    c. empathy maps
    d. needs assessment (techniques for uncovering needs and gathering requirements - e.g., interviews, surveys, ethnographic and contextual enquiry) (See also: SE-Requirements)
    e. journey maps
    f. evaluating the design (See also: HCI-Evaluation)
3. Physical & cognitive characteristics of the user:
    a. physical capabilities that inform interaction design (e.g., color perception, ergonomics)
    b. cognitive models that inform interaction design (e.g., attention, perception and recognition, movement, memory)
    c. topics in social/behavioral psychology (e.g., cognitive biases, change blindness)
4. Designing for diverse user populations: (See also: SEP-IDEA)
    a. how differences (e.g., in race, ability, age, gender, culture, experience, and education) impact user experiences and needs
    b. Internationalization
    c. designing for users from other cultures
    d. cross-cultural design
    e. challenges to effective design evaluation (e.g., sampling, generalization; disability and disabled experiences)

    f. universal design
    g. See also: HCI-Accessibility.
5. Collaboration and communication (See also: AI-SEP 3.e, SE-Teamwork, SEP-Communication 3-5, SPD-Game: 5.d)
    a. understanding the user in a multi-user context
    b. synchronous group communication (e.g., chat rooms, conferencing, online games)
    c. asynchronous group communication (e.g., email, forums, social networks)
    d. social media, social computing, and social network analysis
    e. online collaboration
    f. social coordination and online communities
    g. avatars, characters, and virtual worlds

Illustrative Learning Outcomes:
CS Core:
1. Conduct a user-centered design process that is integrated into a project.

KA Core:
2. Compare and contrast the needs of users with those of designers.
3. Identify the representative users of a design and discuss who else could be impacted by it.
4. Describe empathy and evaluation as elements of the design process.
5. Carry out and document an analysis of users and their needs.
6. Construct a user story from a needs assessment.
7. Redesign an existing solution to a population whose needs differ from those of the initial target population.
8. Contrast the different needs-finding methods for a given design problem.
9. Reflect on whether your design would benefit from low-tech or no-tech components.

Non-Core:
10. Recognize the implications of designing for a multi-user system/context.

HCI-Accountability: Accountability and Responsibility in Design (See also: SE-Tools, SEP-Context, SEP-Ethical-Analysis, SEP-Professional-Ethics, SEP-Privacy, SEP-Communication 7, SEP-Sustainability, SEP-Security, SEP-IDEA, SEC-Foundations)

CS Core:
1. Design impact: sustainability, inclusivity, safety, security, privacy, harm, and disparate impact.
2. Ethics: in design methods and solutions; the role of artificial intelligence; responsibilities for considering stakeholder impact and human factors, role of design to meet user needs.
3. Requirements in design: ownership responsibility, legal frameworks, compliance requirements, consideration beyond immediate user needs, including via iterative reconstruction of problem analysis.

KA Core:
4. Value-sensitive design: identify direct and indirect stakeholders, determine and include diverse stakeholder values and value systems.

5. Persuasion through design: assessing the persuasive content of a design, persuasion as a design goal.

Illustrative Learning Outcomes:
CS Core:

1. Identify and critique the potential impacts of a design on society and relevant communities to address such concerns as sustainability, inclusivity, safety, security, privacy, harm, and disparate impact

KA Core:

2. Identify the potential human factor elements in a design.
3. Identify and understand direct and indirect stakeholders.
4. Develop scenarios that consider the entire lifespan of a design, beyond the immediately planned uses that anticipate direct and indirect stakeholders.
5. Identify and critique the potential factors in a design that impact direct and indirect stakeholders and broader society (e.g., transparency, sustainability of the system, trust, artificial intelligence)
6. Assess the persuasive content of a design and its intent relative to user interests
7. Critique the outcomes of a design given its intent
8. Understand the impact of design decisions

HCI-Accessibility: Accessibility and Inclusive Design (See also: SEP-IDEA, SEP-Security)

CS Core:

1. Background: societal and legal support for and obligations to people with disabilities; accessible design benefits everyone (See also: SEP-IDEA)
2. Techniques: accessibility standards (e.g., Web Content Accessibility Guidelines) (See also: SPD-Web 4)
3. Technologies: features and products that enable accessibility and support inclusive development by designers and engineers
4. Inclusive Design Frameworks: recognizing differences; universal design (See also: SEP-IDEA)

KA Core:

5. Background
    a. demographics and populations (permanent, temporary and situational disability)
    b. international perspectives on disability
    c. attitudes towards people with disabilities
6. Techniques
    a. UX (user experience) design and research
    b. software engineering practices that enable inclusion and accessibility.
7. Technologies: examples of accessibility-enabling features, such as conformance to screen readers
8. Inclusive Design Frameworks: creating inclusive processes such as participatory design; designing for larger impact.

Non-Core:

9. Background

    a. unlearning and questioning
    b. disability studies
10. Technologies: the Return on Investment of inclusion
11. Inclusive Design Frameworks: user-sensitive inclusive design
12. Critical approaches to HCI:
    a. critical race theory in HCI
    b. feminist HCI
    c. critical disability theory.

Illustrative Learning Outcomes:
CS Core:
1. Identify accessibility challenges faced by people with different disabilities, and specify the associated accessible and assistive technologies that address them (See also: AI-Agents 4, 7.a, AI-Robo 13)
2. Identify appropriate inclusive design approaches, such as universal design and ability-based design
3. Identify and demonstrate understanding of software accessibility guidelines
4. Demonstrate recognition of laws and regulations applicable to accessible design

KA Core:
5. Apply inclusive frameworks to design, such as universal design and usability and ability-based design, and demonstrate accessible design of visual, voice-based, and touch-based UIs.
6. Demonstrate understanding of laws and regulations applicable to accessible design
7. Demonstrate understanding of what is appropriate and inappropriate high level of skill during interaction with individuals from diverse populations
8. Analyze web pages and mobile apps for current standards of accessibility

Non-Core:
9. Biases towards disability, race, and gender have historically, either intentionally or unintentionally, informed technology design
    a. find examples
    b. consider how those experiences (learnings?) might inform design.
10. Conceptualize user experience research to identify user needs and generate design insights.

HCI-Evaluation: Evaluating the Design

CS Core:
1. Methods for evaluation with users
    a. formative (e.g. needs-finding and exploratory analysis) and summative assessment (e.g. functionality and usability testing)
    b. elements to evaluate (e.g., utility, efficiency, learnability, user satisfaction)
    c. understanding ethical approval requirements before engaging in user research (See also: SE-Tools, SEP-Ethical-Analysis, SEP-Security, SEP-Privacy)

KA Core:
2. Methods for evaluation with users (See also: SE-Validation)
    a. qualitative methods (qualitative coding and thematic analysis)

    b. quantitative methods (statistical tests)
    c. mixed methods (e.g., observation, think-aloud, interview, survey, experiment)
    d. presentation requirements (e.g., reports, personas)
    e. user-centered testing
    f. heuristic evaluation
    g. challenges and shortcomings to effective evaluation (e.g., sampling, generalization)
3. Study planning
    a. how to set study goals
    b. hypothesis design
    c. approvals from Institutional Research Boards and ethics committees (See also: SEP-Ethical-Analysis, SEP-Security, SEP-Privacy)
    d. how to pre-register a study
    e. within-subjects vs. between-subjects design
4. Implications and impacts of design with respect to the environment, material, society, security, privacy, ethics, and broader impacts. (See also: SEC-Foundations)

Non-Core:

5. Techniques and tools for quantitative analysis
    a. statistical packages
    b. visualization tools
    c. statistical tests (e.g., ANOVA, t-tests, post-hoc analysis, parametric vs non-parametric tests)
    d. data exploration and visual analytics; how to calculate effect size.
6. Data management
    a. data storage and data sharing (open science)
    b. sensitivity and identifiability.

Illustrative Learning Outcomes:
CS Core:

1. Discuss the differences between formative and summative assessment and their role in evaluating design

KA Core:

2. Select appropriate formative or summative evaluation methods at different points throughout the development of a design
3. Discuss the benefits of using both qualitative and quantitative methods for evaluation
4. Evaluate the implications and broader impacts of a given design
5. Plan a usability evaluation for a given user interface, and justify its study goals, hypothesis design, and study design
6. Conduct a usability evaluation of a given user interface and draw defensible conclusions given the study design

Non-Core:

7. Select and run appropriate statistical tests on provided study data to test for significance in the results
8. Pre-register a study design, with planned statistical tests

HCI-Design: System Design (See also: SE-Tools)

CS Core:

1. Prototyping techniques and tools: e.g., low-fidelity prototyping, rapid prototyping, throw-away prototyping, granularity of prototyping
2. Design patterns
    a. iterative design
    b. universal design (See also: SEP-IDEA)
    c. interaction design (e.g., data-driven design, event-driven design)
3. Design constraints
    a. platforms (See also: SPD-Game 3.c)
    b. devices
    c. resources

KA Core:

4. Design patterns and guidelines
    a. software architecture patterns
    b. cross-platform design
    c. synchronization considerations
5. Design processes
    a. participatory design
    b. co-design
    c. double-diamond
    d. convergence and divergence
6. Interaction techniques
    a. input and output vectors (e.g., gesture, pose, touch, voice, force)
    b. graphical user interfaces
    c. controllers
    d. haptics
    e. hardware design
    f. error handling
7. Visual UI design
    a. Color
    b. Layout
    c. Gestalt principles

Non-Core:

8. Immersive environments
    a. virtual reality
    b. augmented reality, mixed reality
    c. XR (which encompasses them)
    d. spatial audio
9. 3D printing and fabrication

10. Asynchronous interaction models
11. Creativity support tools
12. Voice UI designs

Illustrative Learning Outcomes:
CS Core:

1. Propose system designs tailored to a specified appropriate mode of interaction.
2. Follow an iterative design and development process that incorporates
    a. understanding the user
    b. developing an increment
    c. evaluating the increment
    d. feeding those results into a subsequent iteration
3. Explain the impact of changing constraints and design trade offs (e.g., hardware, user, security.) on system design

KA Core:

4. Evaluate architectural design approaches in the context of project goals.
5. Identify synchronization challenges as part of the user experience in distributed environments.
6. Evaluate and compare the privacy implications behind different input techniques for a given scenario
7. Explain the rationale behind a UI design based on visual design principles

Non-Core:

8. Evaluate the privacy implications within a VR/AR/MR scenario

HCI-SEP: Society, Ethics and Professionalism

CS Core:

1. Universal and user-centered design (See also: HCI-User, SEP-IDEA)
2. Accountability (See also: HCI-Accountability)
3. Accessibility and Inclusive Design (See also: SEP-IDEA, SEP-Security)
4. Evaluating the design (See also: HCI-Evaluation)
5. System design (See also: HCI-Design)

KA Core:

6. Participatory and inclusive design processes
7. Evaluating the design: Implications and impacts of design: with respect to the environment, material, society, security, privacy, ethics, and broader impacts (See also: SEC-Foundations, SEP-Privacy)

Illustrative Learning Outcomes:
CS Core:

1. Learning Outcome 1

KA Core:

2. Critique a recent example of a non-inclusive design choice, its societal implications, and propose potential design improvements
3. Evaluating the design: Identify the implications and broader impacts of a given design.

Non-Core:

4. Evaluate the privacy implications within a VR/AR/MR scenario

Professional Dispositions

- Adaptable: An HCI practitioner should be adaptable to address dynamic changes in technology, user needs, and design challenges.
- Meticulous: An HCI practitioner should be meticulous to ensure that their products are both user-friendly and meet the objectives of the design project.
- Empathetic: An HCI practitioner must communicate effectively and create meaningful and enjoyable experiences
- For the user. Team-oriented: The successful HCI practitioner should focus on the success of the team.
- Creative: An HCI practitioner should design solutions that are informed by past practice, the needs of the audience, and HCI fundamentals. Creativity is required to blend these into something that solves the problem appropriately and elegantly.

Math Requirements

Required:

- Basic statistics to support the evaluation and interpretation of results, including central tendency, variability, frequency distribution

, Course Packaging Suggestions

Introduction to HCI for CS majors and minors, to include the following:

- HCI-Accessibility: Accessibility and Inclusive Design: (4 hours)
- HCI-Accountability: Accountability and Responsibility in Design: (2 hours)
- HCI-Design: System Design: (10 hours)
- HCI-Evaluation: Evaluating the Design: (3 hours)
- HCI-SEP: Society, Ethics and Professionalism: (2 hours)
- HCI-User: Understanding the User (7 hours)

Pre-requisites:

- Agile software development

Skill statement: A student who completes this course should be able to describe user-centered design principles and apply them in the context of a small project.

Description: This sample course takes an integrative, project-oriented approach. The students learn HCI principles and apply them in a short, instructor-provided project in weeks 5-6. This motivates students to continue learning new concepts before embarking on a community-engaged final project in which they have to do the requirements analysis, design, implementation, and evaluation using rapid, iterative prototyping.

Suggested weekly topics:

1. Introduction to design (HCI-User, HCI-Design)
2. Thinking, Acting, and Evaluating (HCI-User, HCI-Evaluation, HCI-Design)
3. Memory and Mistakes (HCI-User, HCI-Accessibility,,HCI-Evaluation)
4. Principles and Processes (HCI-Design)

5-6. Integrating Design Processes and Software Development (HCI-Design)

7. Design Thinking and Heuristic Evaluation (HCI-User, HCI-Evaluation)
8. Accessibility (HCI-Accountability, HCI-Accessibility, HCI-SEP)
9. Visual Design and Personas (HCI-User, HCI-Accountability, HCI-Accessibility, HCI-Evaluation, HCI-Design)
10. Final Project: Empathy and Identification (HCI-User, HCI-Accountability, HCI-Accessibility, HCI-Accessibility, HCI-Evaluation, HCI-Design, HCI-SEP)
11. Final Project: Ideation and Low-Fidelity Prototyping (HCI-User, HCI-Evaluation, HCI-Design)

12-13: Final Project Implementation (HCI-Design)

14: Final Project: Testing (HCI-Evaluation)

15: Final Project: Reporting (HCI-User, HCI-Accountability, HCI-Accessibility, HCI-Evaluation, HCI-Design, HCI-SEP)

Introduction to Data Visualization to include the following:

- GIT-B: Visualization: (30 hours)
- GIT-C: Basic Rendering: (10 hours)
- HCI-User: Understanding the User: (3 hours)
- SEP-Privacy, SEP-Ethical-Analysis: (4 hours)

Pre-requisites:

- CS2
- Linear Algebra

Skill statement: A student who completes this course should understand how to select a dataset; ensure the data are accurate and appropriate; design, develop and test a visualization program that depicts the data and is usable.

Advanced Course: Usability Testing

- HCI-User: Understanding the User: (5 hours)
- HCI-Accountability: Accountability and Responsibility in Design: (3 hours)
- HCI-Accessibility: Accessibility and Inclusive Design: (4 hours)
- HCI-Evaluation: Evaluating the Design: (20 hours)
- HCI-Design: System Design: (3 hours)
- HCI-SEP: Society, Ethics and Professionalism: (5 hours)

Pre-requisites:

- Introductory/Foundation courses in HCI
- Research methods, statistics

Description: This project-based course focuses on the formal evaluation of products. Topics include: usability test goal setting, recruitment of appropriate users, design of test tasks, design of the test environment, test plan development and implementation, analysis and interpretation of the results, and documentation and presentation of results and recommendations. Students will understand the techniques, procedures and protocols to apply in various situations for usability testing with users. Students will be able to design an appropriate evaluation plan, effectively conduct the usability test, collect data, and analyze results so that they can suggest improvements.

Suggested topics: planning the usability study, defining goals, study participants, selecting tasks and creating scenarios, deciding how to measure usability, preparing test materials, preparing the test environment, conducting the pilot test, conducting the test, tabulating and analyzing data, recommending changes, communicating the results, preparing the highlight tape, changing the product and the process.

Learning outcomes:

- Design an appropriate test plan
- Recruit appropriate participants
- Conduct a usability test
- Analyze results and recommend changes
- Present results
- Write a report documenting the recommended improvements

Committee

Chair: Susan L. Epstein, Hunter College and The Graduate Center of The City University of New York, New York, USA

Members:

- Sherif Aly, The American University of Cairo, Cairo, Egypt
- Jeremiah Blanchard, University of Florida, Gainesville, FL, USA
- Zoya Bylinskii, Adobe Research, Cambridge, MA, USA
- Paul Gestwicki, Ball State University, Muncie, IN, USA
- Susan Reiser, University of North Carolina at Asheville, Asheville, North Carolina, USA
- Amanda M. Holland-Minkley, Washington and Jefferson College, Washington, PA, USA
- Ajit Narayanan, Google, Mountainview, California, USA
- Nathalie Riche, Microsoft Research Lab, Redmond, WA, USA
- Kristen Shinohara, Rochester Institute of Technology, Rochester, New, York, USA
- Olivier St-Cyr, University of Toronto, Toronto, Canada

Mathematical and Statistical Foundations (MSF)

Preamble

A strong mathematical foundation remains a bedrock of computer science education and infuses the practice of computing whether in developing algorithms, designing systems, modeling real-world phenomena, or computing with data. This Mathematical and Statistical Foundations knowledge (MSF) area – the successor to the ACM CS 2013 curriculum's "Discrete Structures" area – seeks to identify the mathematical and statistical material that undergirds modern computer science. The change of name corresponds to a realization both that the broader name better describes some of the existing topics from 2013 and that some growing areas of computer science, such as artificial intelligence, machine learning, data science, and quantum computing, have continuous mathematics as their foundations too. Because consideration of additional foundational mathematics beyond traditional discrete structures is a substantial matter, the MSF sub-committee included members who have taught courses in continuous mathematics.

The committee considered the following inputs in order to prepare their recommendations:

- A survey distributed to computer science faculty (nearly 600 respondents) across a variety of institutional types and in various countries;
- Math-related data collected from the of survey of industry professionals (865 respondents);
- Math requirement stated by all the knowledge areas in the report;
- Direct input sought from CS theory community; and
- Review of past reports including recent reports on data science (e.g., Park City report) and quantum computing education.

Changes since CS 2013

The breadth of mathematics needed has grown beyond discrete structures to address the mathematical needs of rapidly growing areas such as artificial intelligence, machine learning, robotics, data science, and quantum computing. These areas call for a renewed focus on probability, statistics, and linear algebra, as supported by the faculty survey that asked respondents to rate various mathematical areas in their importance for both an industry career as well as for graduate school; the combined such ratings for probability, statistics, and linear algebra were 98%, 98% and 89% respectively, reflecting a strong consensus in the CS academic community.

Core Hours

Acknowledging some tensions

Several challenges face CS programs when weighing mathematical requirements: (1) many CS majors otherwise engaged in CS, perhaps aiming for a software career, are unenthusiastic about investing in math; (2) institutions such as liberal-arts colleges often limit how many courses a major may require, while others may have common engineering courses that fill up the schedule; and (3) some programs adopt a more pre-professional curricular outlook while others emphasize a more foundational one.

Thus, we are hesitant to recommend an all-encompassing set of mathematical topics as “every CS degree must require.” Instead, we outline two sets of *core* requirements, a minimal “CS-core” set suited to credit-limited majors and a more expansive “KA-core” set to align with technically focused programs. The principle here is that, in light of the additional foundational mathematics needed for AI, data science and quantum computing, programs ought to consider as much as possible from the more expansive KA-core version unless there are sound institutional reasons for alternative requirements.

Knowledge Units	**CS Core**	**KA Core**
Discrete Mathematics	29	11
Probability	11	29
Statistics	10	30
Linear Algebra	5	35
Calculus	0	40
Total	**55**	**200**

Rationale for recommended hours

CS-Core. While some discrete math courses include probability, we highlight its importance so that a minimum number of hours (11) is devoted to probability given its increased importance for AI and data science, and the strong consensus in the academic community based on the survey. Taken together, the total CS core across discrete math and probability (40 hours) is typical of a one-semester 3-credit course. 15 hours are allotted to statistics and linear algebra for basic definitions so that, for example, students should at least be familiar with terms like *mean*, *standard deviation* and *vector*. These could be covered in CS courses. Many programs typically include a broader statistics requirement.

KA-Core. Note that the calculus hours roughly correspond to the typical Calculus-I course now standard across the world. Based on our survey, most programs already require Calculus-I. However, we have left out Calculus-II (an additional 40 hours) and leave it to programs to decide whether Calculus-II should be added to program requirements. Programs could choose to require a more rigorous calculus-based probability or statistics sequence, or non-calculus requiring versions. Similarly, linear algebra can be taught as an applied course without a calculus prerequisite or as a more advanced course.

Knowledge Units

MSF-Discrete: Discrete Mathematics

CS Core:

1. Sets, relations, functions, cardinality

2. Recursive mathematical definitions
3. Proof techniques (induction, proof by contradiction)
4. Permutations, combinations, counting, pigeonhole principle
5. Modular arithmetic
6. Logic: truth tables, connectives (operators), inference rules, formulas, normal forms, simple predicate logic
7. Graphs: basic definitions

Illustrative Learning Outcomes:

1. Sets, Relations, and Functions, Cardinality
    1. *a.* Explain with examples the basic terminology of functions, relations, and sets.
    2. *b.* Perform the operations associated with sets, functions, and relations.
    3. *c.* Relate practical examples to the appropriate set, function, or relation model, and interpret the associated operations and terminology in context.
    4. *d.* Calculate the size of a finite set, including making use of the sum and product rules and inclusion-exclusion principle.
    5. *e.* Explain the difference between finite, countable, and uncountable sets.
2. Recursive mathematical definitions
    1. *a.* Apply recursive definitions of sequences or structures (e.g., Fibonacci numbers, linked lists, parse trees, fractals).
    2. *b.* Formulate inductive proofs of statements about recursive definitions.
    3. *c.* Solve a variety of basic recurrence relations.
    4. *d.* Analyze a problem to determine underlying recurrence relations.
    5. *e.* Given a recursive/iterative code snippet, describe its underlying recurrence relation, hypothesize a closed form for the recurrence relation, and prove the hypothesis correct (probably using induction).
3. Proof Techniques
    1. *a.* Identify the proof technique used in a given proof.
    2. *b.* Outline the basic structure of each proof technique (direct proof, proof by contradiction, and induction) described in this unit.
    3. *c.* Apply each of the proof techniques (direct proof, proof by contradiction, and induction) correctly in the construction of a sound argument.
    4. *d.* Determine which type of proof is best for a given problem.
    5. *e.* Explain the parallels between ideas of mathematical and/or structural induction to recursion and recursively defined structures.
    6. *f.* Explain the relationship between weak and strong induction and give examples of the appropriate use of each.
4. Permutations, combinations, counting
    1. *a.* Apply counting arguments, including sum and product rules, inclusion-exclusion principle and arithmetic/geometric progressions.
    2. *b.* Apply the pigeonhole principle in the context of a formal proof.
    3. *c.* Compute permutations and combinations of a set, and interpret the meaning in the context of the particular application.

    d. 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).

5. Modular arithmetic
    a. Perform computations involving modular arithmetic.
    b. Explain the notion of greatest common divisor, and apply Euclid's algorithm to compute it.
6. Logic
    a. Convert logical statements from informal language to propositional and predicate logic expressions.
    b. Apply formal methods of symbolic propositional and predicate logic, such as calculating validity of formulae, computing normal forms, or negating a logical statement.
    c. Use the rules of inference to construct proofs in propositional and predicate logic.
    d. 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.
    e. 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.
    f. Describe the strengths and limitations of propositional and predicate logic.
    g. Explain what it means for a proof in propositional (or predicate) logic to be valid.
7. Graphs
    a. Illustrate by example the basic terminology of graph theory, and some of the properties and special cases of types of graphs, including trees.
    b. Demonstrate different traversal methods for trees and graphs, including pre-, post-, and in-order traversal of trees, along with breadth-first and depth-first search for graphs.
    c. Model a variety of real-world problems in computer science using appropriate forms of graphs and trees, such as representing a network topology, the organization of a hierarchical file system, or a social network.
    d. Show how concepts from graphs and trees appear in data structures, algorithms, proof techniques (structural induction), and counting.

MSF-Probability: Probability

CS Core:

1. Basic notions: sample spaces, events, probability, conditional probability, Bayes' rule
2. Discrete random variables and distributions
3. Continuous random variables and distributions
4. Expectation, variance, law of large numbers, central limit theorem
5. Conditional distributions and expectation
6. Applications to computing

KA Core:

The recommended topics are the same between CS core and KA-core, but with far more hours, the KA-core can cover these topics in depth and might include more computing-related applications.

Illustrative Learning Outcomes:

1. Basic notions: sample spaces, events, probability, conditional probability, Bayes' rule
    a. Translate a prose description of a probabilistic process into a formal setting of sample spaces, outcome probabilities, and events.
    b. Calculate the probability of simple events.
    c. Determine whether two events are independent.
    d. Compute conditional probabilities, including through applying (and explaining) Bayes' Rule.
2. Discrete random variables and distributions
    a. Define the concept of a random variable and indicator random variable.
    b. Determine whether two random variables are independent.
    c. Identify common discrete distributions (e.g., uniform, Bernoulli, binomial, geometric).
3. Continuous random variables and distributions
    a. Identify common continuous distributions (e.g., uniform, normal, exponential).
    b. Calculate probabilities using cumulative density functions.
4. Expectation, variance, law of large numbers, central limit theorem
    a. Define the concept of expectation and variance of a random variable.
    b. Compute the expected value and variance of simple or common discrete/continuous random variables.
    c. Explain the relevance of the law of large numbers and central limit theorem to probability calculations.
5. Conditional distributions and expectation
    a. Explain the distinction between a joint distribution and a conditional distribution.
    b. Compute conditional distributions from a full distribution, for both discrete and continuous random variables.
    c. Compute conditional expectations for both discrete and continuous random variables.
6. Applications to computing
    a. Describe how probability can be used to model real-life situations or applications, such as predictive text, hash tables, and quantum computation.
    b. Apply probabilistic processes in solving computational problems, such as through randomized algorithms or in security contexts.

MSF-Statistics: Statistics

CS Core:

1. Basic definitions and concepts: populations, samples, measures of central tendency, variance
2. Univariate data: point estimation, confidence intervals

KA Core:

3. Multivariate data: estimation, correlation, regression
4. Data transformation: dimension reduction, smoothing
5. Statistical models and algorithms

Illustrative Learning Outcomes:

CS Core:

1. Basic definitions and concepts: populations, samples, measures of central tendency, variance
    a. Create and interpret frequency tables

    b. Display data graphically and interpret graphs (e.g. histograms)
    c. Recognize, describe and calculate means, medians, quantiles (location of data)
    d. Recognize, describe and calculate variances (spread of data)
2. Univariate data: point estimation, confidence intervals
    a. Formulate maximum likelihood estimation (in linear-Gaussian settings) as a least-squares problem
    b. Calculate maximum likelihood estimates
    c. Calculate maximum a posteriori estimates and make a connection with regularized least squares
    d. Compute confidence intervals as a measure of uncertainty

KA Core:

3. Multivariate data: estimation, correlation, regression
    a. Formulate the multivariate maximum likelihood estimation problem as a least-squares problem
    b. Interpret the geometric properties of maximum likelihood estimates
    c. Derive and calculate the maximum likelihood solution for linear regression
    d. Derive and calculate the maximum a posteriori estimate for linear regression
    e. Implement both maximum likelihood and maximum a posteriori estimates in the context of a polynomial regression problem
    f. Formulate and understand the concept of data correlation (e.g., in 2D)
4. Data transformation: dimension reduction, smoothing
    a. Formulate and derive PCA as a least-squares problem
    b. Geometrically interpret PCA (when solved as a least-squares problem)
    c. Understand when PCA works well (one can relate back to correlated data)
    d. Geometrically interpret the linear regression solution (maximum likelihood)
5. Statistical models and algorithms
    a. Apply PCA to dimensionality reduction problems
    b. Understand the trade-off between compression and reconstruction power
    c. Apply linear regression to curve-fitting problems
    d. Understand the concept of overfitting
    e. Discuss and apply cross-validation in the context of overfitting and model selection (e.g., degree of polynomials in a regression context)

MSF-Linear: Linear Algebra

CS Core:

1. Vectors: definitions, vector operations, geometric interpretation, angles

KA Core:

2. Matrices, matrix-vector equation, geometric interpretation, geometric transformations with matrices
3. Solving equations, row-reduction
4. Linear independence, span, basis
5. Orthogonality, projection, least-squares, orthogonal bases

6. Linear combinations of polynomials, Bezier curves
7. Eigenvectors and eigenvalues
8. Applications to computer science: PCA, SVD, page-rank, graphics

Illustrative Learning Outcomes:
CS Core:

1. Vectors: definitions, vector operations, geometric interpretation, angles
    - *a.* Understand algebraic and geometric representations of vectors in R^n and their operations, including addition, scalar multiplication and dot product
    - *b.* List properties of vectors in R^n
    - *c.* Compute angles between vectors in R^n

KA Core:

2. Matrices, matrix-vector equation, geometric interpretation, geometric transformations with matrices
    - *a.* Perform common matrix operations, such as addition, scalar multiplication, multiplication, and transposition
    - *b.* Relate a matrix to a homogeneous system of linear equations
    - *c.* Recognize when two matrices can be multiplied
    - *d.* Relate various matrix transformations to geometric illustrations
3. Solving equations, row-reduction
    - *a.* Formulate, solve, apply, and interpret properties of linear systems
    - *b.* Perform row operations on a matrix
    - *c.* Relate an augmented matrix to a system of linear equations
    - *d.* Solve linear systems of equations using the language of matrices
    - *e.* Translate word problems into linear equations
    - *f.* Perform Gaussian elimination
4. Linear independence, span, basis
    - *a.* Define subspace of a vector space
    - *b.* List examples of subspaces of a vector space
    - *c.* Recognize and use basic properties of subspaces and vector spaces
    - *d.* Determine whether or not particular subsets of a vector space are subspaces
    - *e.* Discuss the existence of a basis of an abstract vector space
    - *f.* Describe coordinates of a vector relative to a given basis
    - *g.* Determine a basis and the dimension of a finite-dimensional space
    - *h.* Discuss spanning sets for vectors in R^n
    - *i.* Discuss linear independence for vectors in R^n
    - *j.* Define the dimension of a vector space
5. Orthogonality, projection, least-squares, orthogonal bases
    - *a.* Explain the Gram-Schmidt orthogonalization process
    - *b.* Define orthogonal projections
    - *c.* Define orthogonal complements
    - *d.* Compute the orthogonal projection of a vector onto a subspace, given a basis for the subspace
    - *e.* Explain how orthogonal projections relate to least square approximations

6. Linear combinations of polynomials, Bezier curves
    a. Identify polynomials as generalized vectors
    b. Explain linear combinations of basic polynomials
    c. Understand orthogonality for polynomials
    d. Distinguish between basic polynomials and Bernstein polynomials
    e. Apply Bernstein polynomials to Bezier curves
7. Eigenvectors and eigenvalues
    a. Find the eigenvalues and eigenvectors of a matrix
    b. Define eigenvalues and eigenvectors geometrically
    c. Use characteristic polynomials to compute eigenvalues and eigenvectors
    d. Use eigenspaces of matrices, when possible, to diagonalize a matrix
    e. Perform diagonalization of matrices
    f. Explain the significance of eigenvectors and eigenvalues
    g. Find the characteristic polynomial of a matrix
    h. Use eigenvectors to represent a linear transformation with respect to a particularly nice basis
8. Applications to computer science: PCA, SVD, page-rank, graphics
    a. Explain the geometric properties of PCA
    b. Relate PCA to dimensionality reduction
    c. Relate PCA to solving least-squares problems
    d. Relate PCA to solving eigenvector problems
    e. Apply PCA to reducing the dimensionality of a high-dimensional dataset (e.g., images)
    f. Explain the page-rank algorithm and understand how it relates to eigenvector problems
    g. Explain the geometric differences between SVD and PCA
    h. Apply SVD to a concrete example (e.g., movie rankings)

MSF-Calculus

KA Core:

1. Sequences, series, limits
2. Single-variable derivatives: definition, computation rules (chain rule etc), derivatives of important functions, applications
3. Single-variable integration: definition, computation rules, integrals of important functions, fundamental theorem of calculus, definite vs indefinite, applications (including in probability)
4. Parametric and polar representations
5. Taylor series
6. Multivariate calculus: partial derivatives, gradient, chain-rule, vector valued functions, applications to optimization, convexity, global vs local minima
7. ODEs: definition, Euler method, applications to simulation

Note: the calculus topics listed above are aligned with computer science goals rather than with traditional calculus courses. For example, multivariate calculus is often a course by itself but computer science undergraduates only need parts of it for machine learning.

Illustrative Learning Outcomes:

1. Sequences, series, limits
    a. Explain the difference between infinite sets and sequences
    b. Explain the formal definition of a limit
    c. Derive the limit for examples of sequences and series
    d. Explain convergence and divergence
    e. Apply L'Hospital's rule and other approaches to resolving limits
2. Single-variable derivatives: definition, computation rules (chain rule etc), derivatives of important functions, applications
    a. Explain a derivative in terms of limits
    b. Explain derivatives as functions
    c. Perform elementary derivative calculations from limits
    d. Apply sum, product and quotient rules
    e. Work through examples with important functions
3. Single-variable integration: definition, computation rules, integrals of important functions, fundamental theorem of calculus, definite vs indefinite, applications (including in probability)
    a. Explain the definitions of definite and indefinite integrals
    b. Apply integration rules to examples with important functions
    c. Explore the use of the fundamental theorem of calculus
    d. Apply integration to problems
4. Parametric and polar representations
    a. Apply parametric representations of important curves
    b. Apply polar representations
5. Taylor series
    a. Derive Taylor series for some important functions
    b. Apply the Taylor series to approximations
6. Multivariate calculus: partial derivatives, gradient, chain-rule, vector valued functions, applications to optimization, convexity, global vs local minima
    a. Compute partial derivatives and gradients
    b. Work through examples with vector-valued functions with gradient notation
    c. Explain applications to optimization
7. ODEs: definition, Euler method, applications to simulation
    a. Apply the Euler method to integration
    b. Apply the Euler method to a single-variable differential equation
    c. Apply the Euler method to multiple variables in an ODE

Professional Dispositions

We focus on dispositions helpful to students learning mathematics as well as professionals who need to refresh previously learned mathematics or learn new topics.

- ***Growth mindset***. Perhaps the most important of the dispositions, students should be persuaded that anyone can learn mathematics and that success is not based dependent on innate ability.
- ***Practice mindset***. Students should be educated about the nature of "doing" mathematics and learning through practice with problems as opposed to merely listening or observing demonstrations in the classroom.

- ***Delayed gratification***. Most students are likely to learn at least some mathematics from mathematics departments unfamiliar with computing applications; computing departments should acclimate the students to the notion of waiting to see computing applications. Many of the new growth areas such as AI or quantum computing can serve as motivation.
- ***Persistence***. Student perceptions are often driven by frustration with unable to solve hard problems that they see some peers tackle seemingly effortlessly; computing departments should help promote the notion that eventual success through persistence is what matters.

Math Requirements

The intent of this section is to list the likely most important topics that should expected from students entering a computing program, typically corresponding to pre-calculus in high school. We recommend pre-calculus as a prerequisite for discrete mathematics.

Required:

- Algebra and numeracy:
  - Numeracy: numbers, operations, types of numbers, fluency with arithmetic, exponent notation, rough orders of magnitude, fractions and decimals.
  - Algebra: rules of exponents, solving linear or quadratic equations with one or two variables, factoring, algebraic manipulation of expressions with multiple variables.
- Precalculus:
  - Coordinate geometry: distances between points, areas of common shapes
  - Functions: function notation, drawing and interpreting graphs of functions
  - Exponentials and logarithms: a general familiarity with the functions and their graphs
  - Trigonometry: familiarity with basic trigonometric functions and the unit circle

Course Packaging Suggestions

Every department faces constraints in delivering content, which precludes merely requiring a long list of courses covering every single desired topic. These constraints include content-area ownership, faculty size, student preparation, and limits on the number of departmental courses a curriculum can require. We list below some options for mathematical foundations, combinations of which might best fit any particular institution:

- ***Traditional course offerings***. With this approach, a computer science department can require students to take math-department courses in any of the five broad mathematical areas listed above.
- ***A "Continuous Structures" analog of Discrete Structures***. Many computer science departments now offer courses that prepare students mathematically for AI and machine learning. Such courses can combine just enough calculus, optimization, linear algebra and probability; yet others may split linear algebra into its own course. These courses have the advantage of motivating students with computing applications, and including programming as pedagogy for mathematical concepts.
- ***Integration into application courses***. An application course, such as machine learning, can be spread across two courses, with the course sequence including the needed mathematical preparation taught just-in-time, or a single machine learning course can balance preparatory material with new topics. This may have the advantage of mitigating turf issues and helping students see applications immediately after encountering math.

- ***Specific course adaptations***. For nearly a century, physics and engineering needs have driven the structure of calculus, linear algebra, and probability. Computer science departments can collaborate with their colleagues in math departments to restructure math-offered sections in these areas that are driven by computer science applications. For example, calculus could be reorganized to fit the needs of computing programs into two calculus courses, leaving a later third calculus course for engineering and physics students.

Committee

Chair: Rahul Simha, The George Washington University, Washington DC, USA

Members:

- Richard Blumenthal, Regis University, Denver, CO, USA
- Marc Deisenroth, University College London, London, UK
- Michael Goldweber, Denison University, Granville, OH, USA
- David Liben-Nowell, Carleton College, Northfield, MN, USA
- Jodi Tims, Northeastern University, Boston, MA, USA

Networking and Communication (NC)

Preamble

Networking and communication play a central role in interconnected computer systems that are transforming the daily lives of billions of people. The public Internet provides connectivity for networked applications that serve ever-increasing numbers of individuals and organizations around the world. Complementing the public sector, major proprietary networks leverage their global footprints to support cost-effective distributed computing, storage, and content delivery. Advances in satellite networks expand connectivity to rural areas. Device-to-device communication underlies the emerging Internet of things.

This knowledge area deals with key concepts in networking and communication, as well as their representative instantiations in the Internet and other computer networks. Beside the basic principles of switching and layering, the area at its core provides knowledge on naming, addressing, reliability, error control, flow control, congestion control, domain hierarchy, routing, forwarding, modulation, encoding, framing, and access control. The area also covers knowledge units in network security and mobility, such as security threats, countermeasures, device-to-device communication, and multihop wireless networking. In addition to the fundamental principles, the area includes their specific realization in the Internet as well as hands-on skills in implementation of networking and communication concepts. Finally, the area comprises emerging topics such as network virtualization and quantum networking.

As the main learning outcome, learners develop a thorough understanding of the role and operation of networking and communication in networked computer systems. They learn how network structure and communication protocols affect behavior of distributed applications. The area educates on not only key principles but also their specific instantiations in the Internet and equips the student with hands-on implementation skills. While computer-system, networking, and communication technologies are advancing at a fast pace, the gained fundamental knowledge enables the student to readily apply the concepts in new technological settings.

Changes since CS 2013

Compared to the 2013 curricula, the knowledge area broadens its core tier-1 focus from the introduction and networked applications to include reliability support, routing, forwarding, and single-hop communication. Due to the enhanced core, learners acquire a deeper understanding of the impact that networking and communication have on behavior of distributed applications. Reflecting the increased importance of network security, the area adds a respective knowledge unit as a new elective. To track the advancing frontiers in networking and communication knowledge, the area replaces the elective unit on social networking with a new elective unit on emerging topics, such as middleboxes, software defined networks, and quantum networking. Other changes consist of redistributing all topics from the old unit on resource allocation among other units, in order to resolve the unnecessary overlap between the knowledge units in the 2013 curricula.

Core Hours

Knowledge Units	**CS Core**	**KA Core**
Introduction	3	
Networked Applications	4	
Reliability Support		6
Routing And Forwarding		4
Single-Hop Communication		3
Mobility Support		4
Network Security		3
Emerging Topics		4
Total	**7**	**24**

Knowledge Units

NC-Introduction

CS Core:

1. Importance of networking in contemporary computing, and associated challenges. (See also: SEP-Social Context SEP-Privacy and Civil Liberties)
2. Organization of the Internet (e.g. users, Internet Service Providers, autonomous systems, content providers, content delivery networks).
3. Switching techniques (e.g., circuit and packet).
4. Layers and their roles (application, transport, network, datalink, and physical).
5. Layering principles (e.g. encapsulation and hourglass model).
6. Network elements (e.g. routers, switches, hubs, access points, and hosts).
7. Basic queueing concepts (e.g. relationship with latency, congestion, service levels, etc.)

Learning Outcomes:

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

NC-Networked-Applications

CS Core:

1. Naming and address schemes (e.g. DNS, and Uniform Resource Identifiers).
2. Distributed application paradigms (e.g. client/server, peer-to-peer, cloud, edge, and fog). (See also: PDC-Communication, PDC-Coordination)

3. Diversity of networked application demands (e.g. latency, bandwidth, and loss tolerance). (See also: PDC-Communication, SEP-Sustainability)
4. An explanation of at least one application-layer protocol (e.g. HTTP).
5. Interactions with TCP, UDP, and Socket APIs. (See also: PDC-Programs and Execution)

Illustrative Learning Outcomes:

1. Define the principles of naming, addressing, resource location.
2. Analyze the needs of specific networked application demands. SEP-Social Context
3. Describe the details of one application layer protocol. .
4. Implement a simple client-server socket-based application.

NC-Reliability-Support

KA Core:

1. Unreliable delivery (e.g. UDP).
2. Principles of reliability (e.g. delivery without loss, duplication, or out of order).
3. Error control (e.g. retransmission, error correction).
4. Flow control (e.g. stop and wait, window based).
5. Congestion control (e.g. implicit and explicit congestion notification).
6. TCP and performance issues (e.g. Tahoe, Reno, Vegas, Cubic).

Illustrative Learning Outcomes:

1. Describe the operation of reliable delivery protocols.
2. List the factors that affect the performance of reliable delivery protocols.
3. Describe some TCP reliability design issues.
4. Design and implement a simple reliable protocol.

NC-Routing-and-Forwarding

KA Core:

1. Routing paradigms and hierarchy (e.g. intra/inter domain, centralized and decentralized, source routing, virtual circuits, QoS).
2. Forwarding methods (e.g. forwarding tables and matching algorithms).
3. IP and Scalability issues (e.g. NAT, CIDR, BGP, different versions of IP).

Learning Outcomes:

1. Describe various routing paradigms and hierarchies.
2. Describe how packets are forwarded in an IP network.
3. Describe how the Internet tackles scalability challenges. .

NC-Single-Hop-Communication

KA Core:

1. Introduction to modulation, bandwidth, and communication media.
2. Encoding and Framing.
3. Medium Access Control (MAC) (e.g. random access and scheduled access).
4. Ethernet and WiFi.
5. Switching (e.g. spanning trees, VLANS).

6. Local Area Network Topologies (e.g. data center, campus networks).

Illustrative Learning Outcomes:

1. Describe some basic aspects of modulation, bandwidth, and communication media.
2. Describe in detail on a MAC protocol.
3. Demonstrate understanding of encoding and framing solution tradeoffs.
4. Describe details of the implementation of Ethernet
5. Describe how switching works
6. Describe one kind of a LAN topology

NC-Network-Security

KA Core:

1. General intro about security (Threats, vulnerabilities, and countermeasures). (See also: SEP-Security, SEC-Foundations)
2. Network specific threats and attack types (e.g., denial of service, spoofing, sniffing and traffic redirection, man-in-the-middle, message integrity attacks, routing attacks, ransomware, and traffic analysis) (See also: SEC-Foundations)
3. Countermeasures (See also: SEC-Foundations, SEC-Cryptography)
    - Cryptography (e.g. SSL, TLS, symmetric/asymmetric).
    - Architectures for secure networks (e.g., secure channels, secure routing protocols, secure DNS, VPNs, DMZ, Zero Trust Network Access, hyper network security, anonymous communication protocols, isolation)
    - Network monitoring, intrusion detection, firewalls, spoofing and DoS protection, honeypots, tracebacks, BGP Sec, RPKI.

Illustrative Learning Outcomes:

1. Describe some of the threat models of network security.
2. Describe specific network-based countermeasures.
3. Analyze various aspects of network security from a case study.

NC-Mobility

KA Core:

1. Principles of cellular communication (e.g. 4G, 5G).
2. Principles of Wireless LANs (mainly 802.11).
3. Device to device communication.
4. Multihop wireless networks. (e.g. ad hoc networks, opportunistic, delay tolerant).

Illustrative Learning Outcomes:

1. Describe some aspects of cellular communication such as registration.
2. Describe how 802.11 supports mobile users.
3. Describe practical uses of device to device communication, as well as multihop.
4. Describe one type of mobile network such as ad hoc.

NC-Emerging-topics

KA Core:

1. Middleboxes (e.g. filtering, deep packet inspection, load balancing, NAT, CDN).
2. Network Virtualization (e.g. SDN, Data Center Networks).
3. Quantum Networking (e.g. Intro to the domain, teleportation, security, Quantum Internet).
4. Satellite, mmWave, Visible Light.

Illustrative Learning Outcomes:

1. Describe the value of middleboxes in networks.
2. Describe the importance of Software Defined Networks
3. Describe some of the added value achieved by using Quantum Networking

Professional Dispositions

- Meticulous: In meeting being able to design networks and communication systems.
- Collaborative: Working in groups to achieve a common objective.
- Proactive: Anticipating changes in needs and acting upon them.
- Professional: Complying to the needs of the community in a responsible manner.
- Responsive: Acting swiftly to changes in needs.
- Adaptive: Making the required changes happen when needed.

Math Requirements

Required:

- Probability and Statistics
- Discrete Math
- Simple queuing theory concepts.
- Fourier and trigonometric analysis for physical layer.

Course Packaging Suggestions

Coverage of the concepts of networking including but not limited to types of applications used by the network, reliability, routing and forwarding, single hop communication, security, and other emerging topics.

Note: both courses cover the same KU’s but with different allocation of hours for each KU.

Introductory Course:

- NC-Introduction (9 hours)
- NC-Networked Applications (12 hours)

- NC-Reliability Support (6 hours)
- NC-Routing and Forwarding (4 hours)
- NC-Single-Hop Communication (3 hours)
- NC-Network Security (3 hours)
- NC-Mobility Support (3 hours)
- NC-Emerging Topics (2 hours)

Advanced Course:

- NC-Introduction (3 hours)
- NC-Networked Applications (4 hours)
- NC-Reliability Support (8 hours)
- NC-Routing and Forwarding (6 hours)
- NC-Single-Hop Communication (5 hours)
- NC-Network Security (5 hours)
- NC-Mobility Support (5 hours)
- NC-Emerging Topics (6 hours)

Committee

Chair: Sherif G. Aly - The American University in Cairo

Members:

- Khaled Harras: Carnegie Mellon University, Pittsburgh, USA
- Moustafa Youssef: The American University in Cairo, Cairo, Egypt
- Sergey Gorinsky: IMDEA Networks Institute
- Qiao Xiang: Xiamen University, China

Contributor:

- Alex (Xi) Chen: Huawei

Operating Systems (OS)

Preamble

Operating system is the collection of services needed to safely interface the hardware with applications. Core topics focus on the mechanisms and policies needed to virtualize computation, memory, and I/O. Overarching themes that are reused at many levels in computer systems are well illustrated in operating systems (e.g. polling vs interrupts, caching, flexibility costs overhead, similar scheduling approaches to processes, page replacement, etc.). OS should focus on how those concepts apply in other areas of CS - trust boundaries, concurrency, persistence, safe extensibility.

Operating systems remains an important Computer Science Knowledge Area in spite of how OS functions may be redistributed into computer architecture or specialized platforms. A CS student needs to have a clear mental model of how a pipelined instruction executes to how data scope impacts memory location. Students can apply basic OS knowledge to domain-specific architectures (machine learning with GPUs or other parallelized systems, mobile devices, embedded systems, etc.). Since all software must leverage operating systems services, students can reason about the efficiency, required overhead and the tradeoffs inherent to any application or code implementation. The study of basic OS algorithms and approaches provides a context against which students can evaluate more advanced methods. Without an understanding of sandboxing, how programs are loaded into processes, and execution, students are at a disadvantage when understanding or evaluating vulnerabilities to vectors of attack.

Changes since CS 2013

The core of operating systems knowledge from CC2013 has been propagated from CC2023 to the updated knowledge area. Changes from CC2013 include moving of File systems knowledge (now called File Systems API and Implementation) and Device Management to KA Core from elective and Performance and Evaluation knowledge units to the Systems Fundamentals Knowledge area. The addition of persistent data storage and device I/O reflects the impact of file storage and device I/O limitations on the performance (e.g. parallel algorithms, etc.). More advanced topics in File Systems API and Implementation and Device Management were moved to a new Knowledge Unit Advanced File Systems. The Performance and Evaluation knowledge unit moved to Systems Fundamentals with the idea that performance and evaluation approaches for operating systems are mirrored at other levels and are best presented in this context.

Systems programming and creation of platform specific executables are operating systems related topics as they utilize the interface provided by the operating system. These topics are listed as knowledge units within the Foundations of Programming languages (FPL) knowledge area because they are also programming related and would benefit from that context.

Overview

“Role and purpose of operating systems” and “Principles of operating systems” provide a high-level overview of traditional operating systems responsibilities. Required computer architecture mechanisms for safe multitasking and resource management are presented. This provides a basis for application services needed to provide a virtual processing environment. These items are in the CS Core because they enable reasoning on possible security threat vectors and application performance bottlenecks.

“Concurrency” CS Core topics focus on programming paradigms that are needed to share resources within and between operating systems and applications. “Concurrency” KA Core topic provides enough depth into concurrency primitives and solution design patterns so that students can evaluate, design, and implement correct parallelized software components. Although many students may not become operating systems developers, parallel components are widely used in specialized platforms and GPU-based machine learning applications. Non-core topics focus on emerging concepts and examples where there is more integration between architecture, operating systems functions and application software to improve performance and safety.

“Protection and security” CS Core overlaps the dedicated Security Knowledge Area. However, operating systems provide a unique perspective that considers the lower level mechanisms that must be secured for safe system function. “Protection and security” KA Core extends the CS Core topics to operating systems access and control functions that are available to applications and end-users. Non-core focuses on advanced security mechanisms within specific operating systems as well as emerging topics.

“Scheduling”, “Process model”, “Memory Management”, “Device management” and “File systems API and Implementation” KA Core provide depth to the CS Core topics. They provide the basis for virtualization and safe resource management. The placement of these topics in the KA Core does not reduce their importance. It is expected that many of these topics will be covered along with the CS Core topics. Non-core topics focus on emerging topics and provide additional depth to the KA Core topics.

“Society, Ethics and Professionalism” KA Core focuses on open source and life cycle issues. These software engineering issues are not the sole purview of operating systems as they also exist for specialized platforms and applications level knowledge areas.

“Advanced File Systems”, “Virtualization”, “Real-time and Embedded Systems”, and “Fault tolerance” KA Core and Non Core include advanced topics. These topics overlap with “Specialized Platform”, “Architecture”, “Parallel and Distributed Systems” and “Systems Fundamentals” Knowledge Areas.

Core Hours

Knowledge Units	CS Core	KA Core
Role and Purpose of Operating Systems	2	

Principles of Operating System	2	
Concurrency	2	1
Protection and Safety	2	1
Scheduling		1
Society, Ethics and Professionalism (Hours included in SEP		
Process Model		1
Memory Management		2
Device Management		1
File Systems API and Implementation		2
Virtualization		3
Real-time and Embedded Systems		2
Fault Tolerance		3

Knowledge Units

OS-Purpose: Role and purpose of the operating system

CS Core:

1. Operating system as mediator between general purpose hardware and application-specific software Example concepts: Operating system as an abstract virtual machine via an API)
2. Universal operating system functions
   Example concepts:
   a. Interfaces (process, user, device, etc)
   b. Persistence of data
3. Extended and/or specialized operating system functions (Example concepts: Embedded systems, Server types such as file, web, multimedia, boot loaders and boot security)
4. Design issues (e.g. efficiency, robustness, flexibility, portability, security, compatibility, power, safety) Example concepts: Tradeoffs between error checking and performance, flexibility and performance, and security and performance
5. Influences of security, networking, multimedia, parallel and distributed computing
6. Overarching concern of security/protection: Neglecting to consider security at every layer creates an opportunity to inappropriately access resources.
   Example concepts:
   a. Unauthorized access to files on an unencrypted drive can be achieved by moving the media to another computer,

b. Operating systems enforced security can be defeated by infiltrating the boot layer before the operating system is loaded and
c. Process isolation can be subverted by inadequate authorization checking at API boundaries
d. Vulnerabilities in system firmware can provide attack vectors that bypass the operating system entirely
e. Improper isolation of virtual machine memory, computing, and hardware can expose the host system to attacks from guest systems
f. The operating system may need to mitigate exploitation of hardware and firmware vulnerabilities, leading to potential performance reductions (e.g. Spectre and Meltdown mitigations)

7. Exposure of operating systems functions in shells and systems programming (See also: FPL-Scripting)

Illustrative Learning Outcomes:

CS Core:

1. Understand the objectives and functions of modern operating systems
2. Evaluate the design issues in different usage scenarios (e.g. real time OS, mobile, server, etc)
3. Understand the functions of a contemporary operating system with respect to convenience, efficiency, and the ability to evolve
4. Understand how evolution and stability are desirable and mutually antagonistic in operating systems function

OS-Principles: Principles of operating systems

CS Core:

1. Operating system software design and approaches such as Monolithic, Layered, Modular, Micro-kernel models and Unikernel
2. Abstractions, processes, and resources
3. Concept of system calls and links to application program interfaces (APIs)(See also: AR-C: Assembly Level Machine Organization)
   Example concepts:
   a. APIs (Win32, Java, Posix, etc) bridge the gap between highly redundant system calls and functions that are most aligned with the requests an application program would make
   b. Approaches to syscall ABI (Linux "perma-stable" vs. breaking ABI every release).
4. The evolution of the link between hardware architecture and the operating system functions
5. Protection of resources means protecting some machine instructions/functions (See also: AR-C: Assembly Level Machine Organization)
   Example concepts
   a. Applications cannot arbitrarily access memory locations or file storage device addresses
   b. Protection of coprocessors and network devices
6. Leveraging interrupts from hardware level: service routines and implementations (See also: AR-C: Assembly Level Machine Organization)
   Example concepts

   a. Timer interrupts for implementing timeslices
   b. I/O interrupts for putting blocking threads to sleep without polling
7. Concept of user/system state and protection, transition to kernel mode using system calls (See also: AR-C: Assembly Level Machine Organization)
8. Mechanism for invoking of system calls, the corresponding mode and context switch and return from interrupt (See also: AR-C: Assembly Level Machine Organization)
9. Performance costs of context switches and associated cache flushes when performing process switches in Spectre-mitigated environments

Illustrative Learning Outcomes

CS Core:

1. Understand how the application of software design approaches to operating systems design/implementation (e.g. layered, modular, etc) affects the robustness and maintainability of an operating system
2. Categorize system calls by purpose
3. Understand dynamics of invoking a system call (passing parameters, mode change, etc)
4. Evaluate whether a function can be implemented in the application layer or can only be accomplished by system calls
5. Apply OS techniques for isolation, protection and throughput across OS functions (e.g. starvation similarities in process scheduling, disk request scheduling, semaphores, etc) and beyond
6. Understand how the separation into kernel and user mode affects safety and performance.
7. Understand the advantages and disadvantages of using interrupt processing in enabling multiprogramming
8. Analyze for potential vectors of attack via the operating systems and the security features designed to guard against them

OS-Concurrency: Concurrency

CS Core:

1. Thread abstraction relative to concurrency
2. Race conditions, critical regions (role of interrupts if needed)(See also: PDC-A: Programs and Execution)
3. Deadlocks and starvation
4. Multiprocessor issues (spin-locks, reentrancy)
5. Multiprocess concurrency vs. multithreading

KA Core:

6. Thread creation, states, structures(See also: SF-B: Basic Concepts)
7. Thread APIs
8. Deadlocks and starvation (necessary conditions/mitigations)
9. Implementing thread safe code (semaphores, mutex locks, cond vars) (See also: AR-G: Performance and Energy Efficiency, SF-E: Performance Evaluation)
10. Race conditions in shared memory (See also: PDC-A: Programs and Execution)

Non-Core:

11. Managing atomic access to OS objects Example concept: Big kernel lock vs. many small locks vs. lockless data structures like lists

Illustrative Learning Outcomes

CS Core:

1. Understand the advantages and disadvantages of concurrency as inseparable functions within the operating system framework
2. Understand how architecture level implementation results in concurrency problems including race conditions
3. Understand concurrency issues in multiprocessor systems

KA Core:

4. Understand the range of mechanisms that can be employed at the operating system level to realize concurrent systems and describe the benefits of each
5. Understand techniques for achieving synchronization in an operating system (e.g., describe how a semaphore can be implemented using OS primitives) including intra-concurrency control and use of hardware atomics
6. Accurately analyze code to identify race conditions and appropriate solutions for addressing race conditions

OS-Protection: Protection and Safety

CS Core:

1. Overview of operating system security mechanisms (See also: SEC-A: Foundational Security)
2. Attacks and antagonism (scheduling, etc) (See also: SEC-A: Foundational Security)
3. Review of major vulnerabilities in real operating systems (See also: SEC-A: Foundational Security)
4. Operating systems mitigation strategies such as backups (See also: SF-F: System Reliability)

KA Core:

5. Policy/mechanism separation (See also: SEC-F-Security Governance)
6. Security methods and devices (See also: SEC-F-Security Governance)
   Example concepts:
   a. Rings of protection (history from Multics to virtualized x86)
   b. x86_64 rings -1 and -2 (hypervisor and ME/PSP)
7. Protection, access control, and authentication (See also: SEC-F-Security Governance)

Illustrative Learning Outcomes

CS Core:

1. Understand the requirement for protection and security mechanisms in an operating systems
2. List and describe the attack vectors that leverage OS vulnerabilities
3. Understand the mechanisms available in an OS to control access to resources

KA Core:

4. Summarize the features and limitations of an operating system that impact protection and security

OS-Scheduling: Scheduling

KA Core:

1. Preemptive and non-preemptive scheduling
2. Schedulers and policies. Example concepts: First come, first serve, Shortest job first, Priority, Round Robin, and Multilevel **(**See also: SF-C: Resource Allocation and Scheduling)
3. Concepts of SMP/multiprocessor scheduling and cache coherence (See also: AR-C: Assembly Level Machine Organization)
4. Timers (e.g. building many timers out of finite hardware timers) (See also: AR-C: Assembly Level Machine Organization)
5. Fairness and starvation

Non-Core:

6. Subtopics of operating systems such as energy-aware scheduling and real-time scheduling (See also: AR-G: Performance and Energy Efficiency, SPD-Embedded, SPD-Mobile 5-D?-)
7. Cooperative scheduling, such as Linux futexes and userland scheduling

Illustrative Learning Outcomes

KA Core:

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
2. Understand relationships between scheduling algorithms and application domains
3. Understand each types of processor schedulers such as short-term, medium-term, long-term, and I/O
4. Evaluate a problem or solution to determine appropriateness for asymmetric and/or symmetric multiprocessing.
5. Evaluate a problem or solution to determine appropriateness as a processes vs threads
6. Understand the need for preemption and deadline scheduling

Non-Core:

7. Understand the ways that the logic embodied in scheduling algorithms is applicable to other operating systems mechanisms, such as first come first serve or priority to disk I/O, network scheduling, project scheduling, and problems beyond computing

OS-Process: Process Model

KA Core:

1. Processes and threads relative to virtualization-Protected memory, process state, memory isolation, etc
2. Memory footprint/segmentation (stack, heap, etc)(See also: AR-C: Assembly Level Machine Organization)

3. Creating and loading executables and shared libraries(See also: FPL-H: Language Translation and Execution or Systems Interaction)

    Examples:

    a. Dynamic linking, GOT, PLT
    b. Structure of modern executable formats like ELF

4. Dispatching and context switching (See also: AR-C: Assembly Level Machine Organization)
5. Interprocess communication (See also: PDC-B: Communication)

   Example concepts: Shared memory, message passing, signals, environment variables, etc

Illustrative Learning Outcomes

KA Core:

1. Understand how processes and threads use concurrency features to virtualize control
2. Understand reasons for using interrupts, dispatching, and context switching to support concurrency and virtualization in an operating system
3. Understand the different states that a task may pass through and the data structures needed to support the management of many tasks
4. Understand the different ways of allocating memory to tasks, citing the relative merits of each
5. Apply the appropriate interprocess communication mechanism for a specific purpose in a programmed software artifact

OS-Memory: Memory Management

KA Core:

1. Review of physical memory, address translation and memory management hardware(See also: AR-D: Memory Hierarchy)
2. Impact of memory hierarchy including cache concept, cache lookup, etc on operating system mechanisms and policy (See also: AR-D: Memory Hierarchy, SF-D: System Performance)

    Example concepts:

    a. CPU affinity and per-CPU caching is important for cache-friendliness and performance on modern processors

3. Logical and physical addressing, address space virtualization(See also: AR-D: Memory Hierarchy)
4. Concepts of paging, page replacement, thrashing and allocation of pages and frames
5. Allocation/deallocation/storage techniques (algorithms and data structure) performance and flexibility

    Example concepts:

    a. Arenas, slab allocators, free lists, size classes, heterogeneously sized pages (hugepages)

6. Memory caching and cache coherence and the effect of flushing the cache to avoid speculative execution vulnerabilities(See also: AR-F: Functional Organization, AR-D: Memory Hierarchy, SF-D: System Performance)
7. Security mechanisms and concepts in memory mgmt including sandboxing, protection, isolation, and relevant vectors of attack

Non-Core:

8. Virtual Memory: leveraging virtual memory hardware for OS services and efficiency

Illustrative Learning Outcomes

KA Core:

1. Explain memory hierarchy and cost-performance trade-offs
2. Summarize the principles of virtual memory as applied to caching and paging
3. Evaluate the trade-offs in terms of memory size (main memory, cache memory, auxiliary memory) and processor speed
4. Describe the reason for and use of cache memory (performance and proximity, how caches complicate isolation and VM abstraction)
5. Code/Develop efficient programs that consider the effects of page replacement and frame allocation on the performance of a process and the system in which it executes

Non-Core:

6. Explain how hardware is utilized for efficient virtualization

OS-Devices: Device management

KA Core:

1. Buffering strategies (See also: AR-E: Interfacing and Communication)
2. Direct Memory Access and Polled I/O, Memory-mapped I/O Example concept: DMA communication protocols (ring buffers etc)(See also: AR-E: Interfacing and Communication)
3. Historical and contextual - Persistent storage device management (magnetic, SSD, etc)

Non-Core:

4. Device interface abstractions, HALs
5. Device driver purpose, abstraction, implementation and testing challenges
6. High-level fault tolerance in device communication

Illustrative Learning Outcomes

KA Core:

1. Understand architecture level device control implementation and link relevant operating system mechanisms and policy (e.g. Buffering strategies, Direct memory access, etc)
2. Understand OS device management layers and the architecture (device controller, device driver, device abstraction, etc)
3. Understand the relationship between the physical hardware and the virtual devices maintained by the operating system
4. Explain I/O data buffering and describe strategies for implementing it
5. Describe the advantages and disadvantages of direct memory access and discuss the circumstances in which its use is warranted

Non-Core:

6. Describe the complexity and best practices for the creation of device drivers

OS-Files: File Systems API and Implementation

KA Core:

1. Concept of a file including Data, Metadata, Operations and Access-mode
2. File system mounting
3. File access control
4. File sharing
5. Basic file allocation methods including linked, allocation table, etc
6. File system structures comprising file allocation including various directory structures and methods for uniquely identifying files (name, identified or metadata storage location)
7. Allocation/deallocation/storage techniques (algorithms and data structure) impact on performance and flexibility (i.e. Internal and external fragmentation and compaction)
8. Free space management such as using bit tables vs linking
9. Implementation of directories to segment and track file location

Illustrative Learning Outcomes

KA Core:

1. Understand the choices to be made in designing file systems
2. Evaluate different approaches to file organization, recognizing the strengths and weaknesses of each
3. Apply software constructs appropriately given knowledge of the file system implementation

OS-SEP: Society, Ethics and Professionalism

KA Core:

1. Open source in operating systems (See also: SEP-Intellectual Property)
   Example concepts
    a. Identification of vulnerabilities in open source kernels
    b. Open source guest operating systems
    c. Open source host operating systems
    d. Changes in monetization (paid vs free upgrades)
2. End-of-life issues with sunsetting operating systems (See also: SE-XXXXXXX)
   Example concepts: Privacy implications of using proprietary operating systems/operating environments, including telemetry, automated scanning of personal data, built-in advertising, and automatic cloud integration

Illustrative Learning Outcomes

KA Core:

1. Understand advantages and disadvantages of finding and addressing bugs in open source kernels
2. Contextualize history and positive and negative impact of Linux as an open source product
3. List complications with reliance on operating systems past end-of-life

4. Understand differences in finding and addressing bugs for various operating systems payment models

OS-AdvFiles: Advanced File systems

KA Core:

1. File systems: partitioning, mount/unmount, virtual file systems
2. In-depth implementation techniques
3. Memory-mapped files (See also AR-E: Interfacing and Communication)
4. Special-purpose file systems
5. Naming, searching, access, backups
6. Journaling and log-structured file systems (See also SF-F: System Reliability)

Non-Core: (including Emerging topics)

1. Distributed file systems (e.g NAS, OSS, SAN, Cloud, etc)
2. Encrypted file systems
3. Fault tolerance (e.g. fsync and other things databases need to work correctly).

Illustrative Learning Outcomes

KA Core:

1. Understand how hardware developments have led to changes in the priorities for the design and the management of file systems
2. Map file abstractions to a list of relevant devices and interfaces
3. Identify and categorize different mount types
4. Understand specific file systems requirements and the specialize file systems features that meet those requirements
5. Understand the use of journaling and how log-structured file systems enhance fault tolerance

Non-Core:

6. Understand purpose and complexity of distributed file systems
7. List examples of distributed file systems protocols
8. Understand mechanisms in file systems to improve fault tolerance

OS-Virtualization: Virtualization

KA Core:

1. Using virtualization and isolation to achieve protection and predictable performance (See also: SF-D-System Performance)
2. Advanced paging and virtual memory
3. Virtual file systems and virtual devices
4. Containers (See also: SF-D-System Performance)

Example concepts: Emphasizing that containers are NOT virtual machines, since they do not contain their own operating systems [where operating system is pedantically defined as the kernel]

5. Thrashing

a. Popek and Goldberg requirements for recursively virtualizable systems

Non-core:

6. Types of virtualization (including Hardware/Software, OS, Server, Service, Network) (See also: SF-D-System Performance)
7. Portable virtualization; emulation vs. isolation (See also: SF-D-System Performance)
8. Cost of virtualization (See also: SF-D-System Performance, SF-E: Performance Evaluation)
9. VM and container escapes, dangers from a security perspective (See also: SF-D-System Performance, SEC-Engineering)
10. Hypervisors- hardware virtual machine extensions
    Example concepts:
    a. Hypervisor monitor w/o a host operating system
    b. Host OS with kernel support for loading guests, e.g. QEMU KVM

Illustrative Learning Outcomes

KA Core:

1. Understand how hardware architecture provides support and efficiencies for virtualization
2. Understand difference between emulation and isolation
3. Evaluate virtualization trade-offs

Non-Core:

4. Understand hypervisors and the need for them in conjunction with different types of hypervisors
    a. Dynamic recompilation as an intermediary between full emulation and use of hardware hypervisor extensions on non-virtualizable ISAs whenever the guest and host system architectures match

OS-Real-time: Real-time/embedded

KA Core:

1. Process and task scheduling
2. Deadlines and real-time issues (See also: SPD-Embedded)
3. Low-latency/soft real-time" vs "hard real time" (See also: SPD-Embedded, FPL-S: Embedded Computing and Hardware Interface)

Non-Core:

4. Memory/disk management requirements in a real-time environment
5. Failures, risks, and recovery
6. Special concerns in real-time systems (safety)

Illustrative Learning Outcomes

KA Core:

1. Understand what makes a system a real-time system
2. Understand latency and its sources in software systems and its characteristics.
3. Understand special concerns that real-time systems present, including risk, and how these concerns are addressed

Non-Core:

4. Understand specific real time operating systems features and mechanisms

OS-Faults: Fault tolerance

KA Core:

1. Reliable and available systems (See also: SF-Reliability 1)
2. Software and hardware approaches to address tolerance (RAID) (See also: SF-Reliability)

Non-Core:

3. Spatial and temporal redundancy (See also: SF-Reliability 2)
4. Methods used to implement fault tolerance (See also: SF-Reliability 2,3)
5. Error identification and correction mechanisms (See also: AR-Memory)
   a. Checksumming of volatile memory in RAM
6. File system consistency check and recovery
7. Journaling and log-structured file systems (See also: SF-Reliability5)
8. Use-cases for fault-tolerance (databases, safety-critical) (See also: SF-Reliability1)
9. Examples of OS mechanisms for detection, recovery, restart to implement fault tolerance, use of these techniques for the OS's own services (See also: SF-Reliability)

Illustrative Learning Outcomes

KA Core:

3. Understand how operating system can facilitate fault tolerance, reliability, and availability
4. Understand the range of methods for implementing fault tolerance in an operating system
5. Understand how an operating system can continue functioning after a fault occurs
6. Understand the performance and flexibility trade offs that impact using fault tolerance

Non-Core:

7. Describe operating systems fault tolerance issues and mechanisms in detail

Professional Dispositions

- **Proactively considers** the implications for security and performance of decisions
- **Meticulously** considers implication of OS mechanisms on any project

Math Requirements

Required:

- Discrete math

Course Packaging Suggestions

Introductory Course to include the following:

- OS-Purpose: Role and Purpose of Operating Systems- 3 hours
- OS-Principles: Principles of Operating Systems- 3 hours
- OS-Concurrency: Concurrency- 7 hours
- OS-Scheduling: Scheduling- 3 hours
- OS-Process: Process Model- 3 hours
- OS-Memory: Memory Management- 4 hours
- OS-Protection: Protect and Safety- 4 hours
- OS-Devices: Device Management- 2 hours
- OS-Files: File Systems API and Implementation- 2 hours
- OS-Virtualization: Virtualization- 3 hours
- OS-AdvFiles: Advanced File Systems- 2 hours
- OS-Real-time: Real-time and Embedded Systems- 1 hours
- OS-Faults: Fault Tolerance- 1 hours
- OS-SEP: Social, Ethical and Professional topics- 4 hours

Pre-requisites:

- Assembly Level Machine Organization from Architecture
- Memory Management from Architecture
- Software Reliability from Architecture
- Interfacing and Communication from Architecture
- Functional Organization from Architecture

Skill statement: A student who completes this course should understand the impact and implications of operating system resource management in terms of performance and security. A student should understand and implement interprocess communication mechanisms safely. A student should differentiate between the use and evaluation of open source and/or proprietary operating systems. A student should understand virtualization as a feature of safe modern operating systems implementation.

Committee

Chair: Monica D. Anderson, University of Alabama, Tuscaloosa, AL, USA

Members:

- Renzo Davoli
- Avi Silberschatz
- Marcelo Pias, Federal University of Rio Grande (FURG), Brazil
- Mikey Goldweber, Xavier University, Cincinnati, USA
- Qiao Xiang, Xiamen University, China

Parallel and Distributed Computing (PDC)

Preamble

Parallel and distributed programming arranges and controls multiple computations occurring at the same time across different places. The ubiquity of parallelism and distribution are inevitable consequences of increasing numbers of gates in processors, processors in computers, and computers everywhere that may be used to improve performance compared to sequential programs, while also coping with the intrinsic interconnectedness of the world, and the possibility that some components or connections fail or misbehave. Parallel and distributed programming remove the restrictions of sequential programming that require computational steps to occur in a serial order in a single place, revealing further distinctions, techniques, and analyses applying at each layer of computing systems.

In most conventional usage, "parallel" programming focuses on arranging that multiple activities co-occur, "distributed" programming focuses on arranging that activities occur in different places, and "concurrent" programming focuses on interactions of ongoing activities with each other and the environment. However, all three terms may apply in most contexts. Parallelism generally implies some form of distribution because multiple activities occurring without sequential ordering constraints happen in multiple physical places (unless relying on context-switching schedulers or quantum effects). And conversely, actions in different places need not bear any particular sequential ordering with respect to each other in the absence of communication constraints..

PDC has evolved from a diverse set of advanced topics into a central body of knowledge and practice, permeating almost every other aspect of computing. Growth of the field has occurred irregularly across different subfields of computing, sometimes with different goals, terminology, and practices, masking the considerable overlap of basic ideas and skills that are the main focus of this KA. Nearly every problem with a sequential solution also admits parallel and/or distributed solutions; additional problems and solutions arise only in the context of existing concurrency. And nearly every application domain of parallel and distributed computing is a well-developed area of study and/or engineering too large to enumerate.

Overview

The PDC KA is divided into five KUs, each with CS-Core and KA-core components that extend but do not overlap CS Core coverage that appears in other KAs.. They cover: The nature of parallel and distributed **Programs** and their execution; **Communication** (via channels, memory, or shared data stores), **Coordination** among parallel activities to achieve common outcomes; **Evaluation** with respect to specifications, and **Algorithms** across multiple application domains..

CS Core topics span approaches to parallel and distributed computing, but restrict coverage to those applying to nearly all of them. Learning Outcomes include developing small programs (in a choice of several styles) with multiple activities and analyzing basic properties. The topics and hours do not include coverage of particular languages, tools, frameworks, systems, and platforms needed as a basis for implementing and evaluating concepts and skills. They also avoid reliance on specifics that may vary widely (for example GPU programming vs cloud container deployment scripts), Prerequisites for PDC CS Core coverage include::

- SDF-Fundamentals. programs vs executions, specifications vs implementations, variables, arrays, sequential control flow, procedural abstraction and invocation, IO.
- SF-Overview: Layered systems, State machines, Reliability
- AR-Assembly, AR-Memory: Von Neumann architecture, Memory hierarchy
- MSF-Discrete: Logic, discrete structures including directed graphs.

Additionally, FPL (Foundations of Programming Languages) may be treated as a prerequisite, depending on other curriculum choices. The PDC CS Core requires familiarity with languages and platforms that enable expression of basic parallel and distributed programs. These need not be included in SDF or SF but are required in FPL-PDC, which additionally covers their use in PDC constructions. Also, PDC includes definitions of Safety, Liveness, and related concepts that are covered with respect to language properties and semantics in FPL. Similarly, the PDC CS Core includes concepts that are further developed in the context of network protocols in NC (Networking and Communication), OS (Operating Systems) and SEC (Security), that could be covered in any order..

KA Core topics in each unit are of the form “One or more of the following” for *a la carte* topics extending associated core topics. Any selection of KA-core topics meeting the KA Core hour requirement constitutes fulfillment of the KA Core. These permit variation in coverage depending on the focus of any given course. See below for examples. Depth of coverage of any KA Core subtopic is expected to vary according to course goals. For example, shared-memory coordination is a central topic in multicore programming, but much less so in most heterogeneous systems, and conversely for bulk data transfer. Similarly, fault tolerance is central to the design of distributed information systems, but much less so in most data-parallel applications.

Changes since CS 2013

The PDC KA has been refactored to focus on commonalities across different forms of parallel and distributed computing, also enabling more flexibility in KA-core coverage, with more guidance on coverage options.

Core Hours

Knowledge Units	**CS Core hours**	**KA Core hours**
Programs	2	2
Communication	2	6
Coordination	2	6
Evaluation	1	3

Algorithms	2	9
Total	**9**	**26**

Knowledge Units

PDC-Programs

CS Core:

1. Fundamental concepts
    a. Ordering
        i. Declarative parallelism: Determining which actions may be performed in parallel, at the level of instructions, functions, closures, composite actions, sessions, tasks, services
        ii. Defining order: happens-before relations, series/parallel directed acyclic graphs representing programs
        iii. Independence: determining when ordering doesn’t matter, in terms of commutativity, dependencies, preconditions
        iv. Ensuring ordering among otherwise parallel actions when necessary, for example locking, safe publication; and orderings imposed by communication: sending a message happens before receiving it
        v. Nondeterministic execution of unordered actions
    b. Places
        i. Devices executing actions include hardware components, remote hosts (See also AR-IO)
        ii. One device may time-slice or otherwise emulate multiple parallel actions by fewer processors (See also OS-Scheduling)
        iii. May include external, uncontrolled devices, hosts, and human users
    c. Deployment
        i. Arranging actions be performed (eventually) at places, with options ranging from from hardwiring to configuration scripts, or reliance on automated provisioning and management by platforms
        ii. Establishing communication and resource management (See also SF-Resources)
        iii. Naming or identifying actions as parties (for example thread IDs)
    d. Consistency
        i. Agreement among parties about values and predicates
        ii. Races, atomicity, consensus
        iii. Tradeoffs of consistency vs progress in decentralized systems
    e. Faults
        i. Handling failures in parties or communication, Including (Byzantine) misbehavior due to untrusted parties and protocols
        ii. Degree of fault tolerance and reliability may be a design choice
2. Programming new activities
    a. The first step of PDC-Coordination techniques. expressed differently across languages, platforms, contexts

    b. Procedural: Enabling multiple actions to start at a given program point; for example, starting new threads, possibly scoping or otherwise organizing them in possibly-hierarchical groups
    c. Reactive: Enabling upon an event by installing an event handler, with less control of when actions begin or end
    d. Dependent: Enabling upon completion of others; for example, sequencing sets of parallel actions

KA Core:

3. Mappings and mechanisms across layered systems. **One or more of:**
    a. CPU data- and instruction-level- parallelism (See also AR-Organization)
    b. SIMD and heterogeneous data parallelism (See also AR-Heterogeneity)
    c. Multicore scheduled concurrency, tasks, actors (See also OS-Scheduling)
    d. Clusters, clouds; elastic provisioning (See also SPD-Common)
    e. Networked distributed systems (See also NC-Networked-Applications)
    f. Emerging technologies such as quantum computing and molecular computing

Illustrative Learning Outcomes

CS Core:

1. Graphically show (as a dag) how to parallelize a compound numerical expression; for example a = (b+c) * (d + e).
2. Explain why the concepts of consistency and fault tolerance do not arise in purely sequential programs

KA Core:

3. Write a function that efficiently counts events such as networking packet receptions
4. Write a filter/map/reduce program in multiple styles
5. Write a service that creates a thread (or other procedural form of activation) to return a requested web page to each new client.

PDC-Communication

CS Core:

1. Media
    a. Varieties: channels (message passing or IO), shared memory, heterogeneous, data stores
    b. Reliance on the availability and nature of underlying hardware, connectivity, and protocols; language support, emulation (See also AR)
2. Channels
    a. Explicit party-to-party communication; naming channels
    b. APIs: sockets (See also NC-Introduction), architectural and language-based constructs
    c. IO channel APIs
3. Memory
    a. Parties directly communicate only with memory at given addresses, with extensions to heterogeneous memory supporting multiple memory stores with explicit data transfer across them; for example, GPU local and shared memory, DMA
    b. Consistency: Bitwise atomicity limits, coherence, local ordering

    c. Memory hierarchies: Multiple layers of sharing domains, scopes and caches; locality: latency, false-sharing

4. Data Stores
    a. Cooperatively maintained structured data implementing maps and related ADTs
    b. Varieties: Owned, shared, sharded, replicated, immutable, versioned
5. Programming with communication
    a. Using channel, socket, and/or remote procedure call APIs
    b. Using shared memory constructs in a given language

KA Core:

6. Properties and Extensions. **One or more of**:
    a. Media
        i. Topologies: Unicast, Multicast, Mailboxes, Switches; Routing via hardware and software interconnection networks
        ii. Concurrency properties: Ordering, consistency, idempotency, overlapping communication with computation
        iii. Performance properties: Latency, bandwidth (throughput) contention (congestion), responsiveness (liveness), reliability (error and drop rates), protocol-based progress (acks, timeouts, mediation)
        iv. Security properties: integrity, privacy, authentication, authorization (See also SEC)
        v. Data formats
        vi. Applications of Queuing Theory to model and predict performance
    b. Channels
        i. Policies: Endpoints, Sessions, Buffering, Saturation response (waiting vs dropping), Rate control
        ii. Program control for sending (usually procedural) vs receiving.(usually reactive or RPC-based)
        iii. Formats, marshaling, validation, encryption, compressIon
        iv. Multiplexing and demultiplexing many relatively slow IO devices or parties; completion-based and scheduler-based techniques; async-await, select and polling APIs.
        v. Formalization and analysis; for example using CSP
    c. Memory
        i. Memory models: sequential and release/acquire consistency
        ii. Memory management; including reclamation of shared data; reference counts and alternatives
        iii. Bulk data placement and transfer; reducing message traffic and improving locality; overlapping data transfer and computation; impact of data layout such as array-of-structs vs struct-of-arrays
        iv. Emulating shared memory: distributed shared memory, RDMA
    d. Data Stores
        i. Consistency: atomicity, linearizability, transactionality, coherence, causal ordering, conflict resolution, eventual consistency, blockchains,
        ii. Faults, partitioning, and partial failures; voting; protocols such as Paxos and Raft

iii. Design tradeoffs among consistency, availability, partition (fault) tolerance; impossibility of meeting all at once
iv. Security and trust: Byzantine failures, proof of work and alternatives

Illustrative Learning Outcomes

CS Core:

1. Explain the similarities and differences among: (1) Party A sends a message on channel X with contents 1 received by party B (2) A sets shared variable X to 1, read by B (3) A sets "X=1' in a distributed shared map accessible by B.
2. Write a producer-consumer program in which one component generates numbers, and another computes their average. Measure speedups when the numbers are small scalars versus large multi-precision values.

KA Core:

3. Write a program that distributes different segments of a data set to multiple workers, and collects results (for the simplest example, summing segments of an array).
4. Write a parallel program that requests data from multiple sites, and summarizes them using some form of reduction
5. Compare the performance of buffered versus unbuffered versions of a producer-consumer program
6. Determine whether a given communication scheme provides sufficient security properties for a given usage
7. Give an example of an ordering of accesses among concurrent activities (e.g., program with a data race) that is not sequentially consistent.
8. Give an example of a scenario in which blocking message sends can deadlock.
9. Describe at least one design technique for avoiding liveness failures in programs using multiple locks
10. Write a program that illustrates memory-access or message reordering.
11. Describe the relative merits of optimistic versus conservative concurrency control under different rates of contention among updates.
12. Give an example of a scenario in which an attempted optimistic update may never complete.
13. Modify a concurrent system to use a more scalable, reliable or available data store
14. Using an existing platform supporting replicated data stores, write a program that maintains a key-value mapping even when one or more hosts fail.

PDC-Coordination

CS Core:

1. Dependencies
   a. Initiation or progress of one activity may be dependent on other activities, so as to avoid race conditions, ensure termination, or meet other requirements
   b. Ensuring progress by avoiding dependency cycles, using monotonic conditions, removing inessential dependencies
2. Control constructs
   a. Completion-based: Barriers, joins
   b. Data-enabled: Queues, Produce-Consumer designs
   c. Condition-based: Polling, retrying, backoffs, helping, suspension, signaling, timeouts

    d. Reactive: enabling and triggering continuations
3. Atomicity
    a. Atomic instructions, enforced local access orderings
    b. Locks and mutual exclusion; lock granularity
    c. Deadlock avoidance: ordering, coarsening, randomized retries; encapsulation via lock managers
    d. Common errors: failing to lock or unlock when necessary, holding locks while invoking unknown operations
    e. Avoiding locks: replication, read-only, ownership, and nonblocking constructions
4. Programming with coordination
    a. Controlling termination
    b. Using locks, barriers, and other synchronizers in a given language; maintaining liveness without introducing races

KA Core:

5. Properties and extensions. **One or more of:**
    a. Progress
        i. Properties including lock-free, wait-free, fairness, priority scheduling; interactions with consistency, reliability
        ii. Performance: contention, granularity, convoying, scaling
        iii. Non-blocking data structures and algorithms
    b. Atomicity
        i. Ownership and resource control
        ii. Lock variants and alternatives: sequence locks, read-write locks; RCU, reentrancy; tickets; controlling spinning versus blocking
        iii. Transaction-based control: Optimistic and conservative
        iv. Distributed locking: reliability
    c. Interaction with other forms of program control
        i. Alternatives to barriers: Clocks; Counters, Virtual clocks; Dataflow and continuations; Futures and RPC; Consensus-based, Gathering results with reducers and collectors
        ii. Speculation, selection, cancellation; observability and security consequences
        iii. Resource-based: Semaphores and condition variables
        iv. Control flow: Scheduling computations, Series-parallel loops with (possibly elected) leaders, Pipelines and Streams, nested parallelism.
        v. Exceptions and failures. Handlers, detection, timeouts, fault tolerance, voting

Illustrative Learning Outcomes

CS Core:

1. Show how to avoid or repair a race error in a given program
2. Write a program that correctly terminates when all of a set of concurrent tasks have completed.

KA Core:

3. Write a function that efficiently counts events such as sensor inputs or networking packet receptions
4. Write a filter/map/reduce program in multiple styles
5. Write a program in which the termination of one set of parallel actions is followed by another

6. Write a program that speculatively searches for a solution by multiple activities, terminating others when one is found.
7. Write a program in which a numerical exception (such as divide by zero) in one activity causes termination of others
8. Write a program for multiple parties to agree upon the current time of day; discuss its limitations compared to protocols such as NTP
9. Write a service that creates a thread (or other procedural form of activation) to return a requested web page to each new client

PDC-Evaluation:

CS Core:

1. Safety and liveness requirements (See also FPL-PDC:1)
    a. Temporal logic constructs to express “always” and “eventually”
2. Identifying, testing for, and repairing violations
    a. Common forms of errors: failure to ensure necessary ordering (race errors), atomicity (including check-then-act errors), or termination (livelock)..
3. Performance requirements
    a. Metrics for throughput, responsiveness, latency, availability, energy consumption, scalability, resource usage, communication costs, waiting and rate control, fairness; service level agreements (See also SF-Performance)
4. Performance impacts of design and implementation choices
    a. Granularity, overhead, energy consumption, and scalability limitations (See also SEP: Sustainability)
    b. Estimating scalability limitations, for example using Amdahl and Gustafson laws (See also SF-Evaluation)

KA Core:

5. Methods and tools. **One or more of:**
    a. Formal Specification
        i. Extensions of sequential requirements such as linearizability; protocol, session, and transactional specifications
        ii. Use of tools such as UML, TLA, program logics
        iii. Security: safety and liveness in the presence of hostile or buggy behaviors by other parties; required properties of communication mechanisms (for example lack of cross-layer leakage), input screening, rate limiting
    b. Static Analysis
        i. Applied to correctness, throughput, latency, resources, energy
        ii. dag model analysis of algorithmic efficiency (work, span, critical paths)
    c. Empirical Evaluation
        i. Testing and debugging; tools such as race detectors, fuzzers, lock dependency checkers, unit/stress/torture tests, visualizations, continuous integration, continuous deployment, and test generators,
        ii. Measuring and comparing throughput, overhead, waiting, contention, communication, data movement, locality, resource usage, behavior in the presence of too many events, clients, threads.
    d. Application domain specific analyses and evaluation techniques

Illustrative Learning Outcomes

CS Core:

1. Revise a specification to enable parallelism and distribution without violating other essential properties or features
2. Explain how concurrent notions of safety and liveness extend their sequential counterparts
3. Specify a set of invariants that must hold at each bulk-parallel step of a computation
4. Write a test program that can reveal a data race error; for example, missing an update when two activities both try to increment a variable.
5. In a given context, explain the extent to which introducing parallelism in an otherwise sequential program would be expected to improve throughput and/or reduce latency, and how it may impact energy efficiency
6. Show how scaling and efficiency change for sample problems without and with the assumption of problem size changing with the number of processors; further explain whether and how scalability would change under relaxations of sequential dependencies.

KA Core:

7. Specify and measure behavior when a service is requested by too many clients
8. Identify and repair a performance problem due to sequential bottlenecks
9. Empirically compare throughput of two implementations of a common design (perhaps using an existing test harness framework).
10. Identify and repair a performance problem due to communication or data latency
11. Identify and repair a performance problem due to communication or data latency
12. Identify and repair a performance problem due to resource management overhead
13. Identify and repair a reliability or availability problem

PDC-Algorithms

CS Core:

1. Expressing and implementing algorithms (See also FPL-PDC)
    a. Implementing concepts in given languages and frameworks to initiate activities (for example threads), use shared memory constructs, and channel, socket, and/or remote procedure call APIs.
    b. Basic examples: map/reduce
2. Survey of common application domains
 (with reference to the following table)

Category	Typical Execution agents	Typical Communication mechanisms	Typical Algorithmic domains	Typical Engineering goals
MultiCore:	Threads	Shared memory, Atomics, locks	Resource management, data processing	throughput, latency, energy

Reactive	Handlers, threads	IO Channels	Services, real-time	latency
Data parallel	GPU, SIMD, accelerators, hybrid	Heterogeneous memory	Linear algebra, graphics, data analysis	throughput, energy
Cluster	Managed hosts	Sockets, channels	Simulation, data analysis	throughput
Cloud	Provisioned hosts	Service APIs	Web applications	scalability
Open Distributed	Autonomous hosts	Sockets, Data stores	Fault tolerant data stores and services	reliability

KA Core:

3. Algorithmic Domains. **One of more of**:
    a. Linear Algebra: Vector and Matrix operations, numerical precision/stability, applications in data analytics and machine learning
    b. Data processing: sorting, searching and retrieval, concurrent data structures
    c. Graphs, search, and combinatorics: Marking, edge-parallelization, bounding, speculation, network-based analytics
    d. Modeling and simulation: differential equations; randomization, N-body problems, genetic algorithms
    e. Computational Logic: SAT, concurrent logic programming
    f. Graphics and computational geometry: Transforms, rendering, ray-tracing
    g. Resource Management: Allocating, placing, recycling and scheduling processors, memory, channels, and hosts. Exclusive vs shared resources. Static, dynamic and elastic algorithms; Real-time constraints; Batching, prioritization, partitioning, Decentralization via work-stealing and related techniques;
    h. Services: Implementing Web APIs, Electronic currency, transaction systems, multiplayer games.

Illustrative Learning Outcomes

CS Core:

1. Implement a parallel/distributed component based on a known algorithm
2. Write a data-parallel program that for example computes the average of an array of numbers.
3. Extend an event-driven sequential program by establishing a new activity in an event handler (for example a new thread in a GUI action handler).
4. Improve the performance of a sequential component by introducing parallelism and/or distribution
5. Choose among different parallel/distributed designs for components of a given system

KA Core:

6. Design, implement, analyze, and evaluate a component or application for X operating in a given context, where X is in one of the listed domains; for example a genetic algorithm for factory floor design.
7. Critique the design and implementation of an existing component or application, or one developed by classmates
8. Compare the performance and energy efficiency of multiple implementations of a similar design; for example multicore versus clustered versus GPU.

Professional Dispositions

- Meticulous: Attention to detail is essential when applying constructs with non-obvious correctness conditions.
- Persistent: Developers must be tolerant of the common need to revise initial approaches when solutions are not self-evident

Math Requirements

- CS Core: Logic, discrete structures including directed graphs.
- KA-Core: Math underlying topics in linear algebra, differential equations

Course Packaging Suggestions

Parallel Computing

- PDC-A: Programs and Execution (4 hours)
- PDC-B: Communication (6 hours)
- PDC-C: Coordination (6 hours)
- PDC-D: Software Engineering (4 hours)
- PDC-E: Algorithms and Application Domains (6 hours)

Distributed Computing

- PDC-A: Programs and Execution (4 hours)
- PDC-B: Communication (3 hours)
- PDC-C: Coordination (3 hours)
- PDC-D: Software Engineering (3 hours)
- PDC-E: Algorithms and Application Domains (6 hours)

High-performance Computing

HPC no prerequisite:

- PDC-A: Programs and Execution (4 hour)
- PDC-B: Communication (6 hours)
- PDC-C: Coordination (6 hours)
- PDC-D: Software Engineering (5 hours)
- PDC-E: Algorithms and Application Domains (11 hours)

HPC with Parallel Computing as prerequisites:

- PDC-A: Programs and Execution (1 hour)
- PDC-B: Communication (2 hours)
- PDC-C: Coordination (2 hours)
- PDC-D: Software Engineering (2 hours)
- PDC-E: Algorithms and Application Domains (6 hours)

More extensive examples and guidance for courses focusing on HPC are provided by the NSF/IEEE-TCPP Curriculum Initiative on Parallel and Distributed Computing (http://tcpp.cs.gsu.edu/curriculum/).

Committee

Chair: Doug Lea, State University of New York at Oswego, Oswego, USA

Members:

- Sherif Aly, American University of Cairo, Cairo, Egypt
- Michael Oudshoorn, High Point University, High Point, NC, USA
- Qiao Xiang, Xiamen University, China
- Dan Grossman, University of Washington, Seattle, USA
- Sebastian Burckhardt, Microsoft Research
- Vivek Sarkar, Georgia Tech, Atlanta, USA
- Maurice Herlihy, Brown University, Providence, USA
- Sheikh Ghafoor, Tennessee Tech, USA
- Chip Weems, University of Massachusetts, Amherst, USA

Contributors:

- Paul McKenney, Meta, Beaverton, OR, USA
- Peter Buhr, University of Waterloo, Waterloo, Ontario, Canada

Software Development Fundamentals (SDF)

Preamble

Fluency in the process of software development is fundamental to the study of computer science. In order to use computers to solve problems most effectively, students must be competent at reading and writing programs. Beyond programming skills, however, they must be able to select and use appropriate data structures and algorithms, and use modern development and testing tools.

The SDF knowledge area brings together fundamental concepts and skills related to software development, focusing on concepts and skills that should be taught early in a computer science program, typically in the first year. This includes fundamental programming concepts and their effective use in writing programs, use of fundamental data structures which may be provided by the programming language, basics of programming practices for writing good quality programs, reading and understanding programs, and some understanding of the impact of algorithms on the performance of the programs. The 43 hours of material in this knowledge area may be augmented with core material from other knowledge areas as a student progresses to mid- and upper-level courses.

This knowledge area assumes a contemporary programming language with good built-in support for common data types including associative data types like dictionaries/maps as the vehicle for introducing students to programming (e.g. Python, Java). However, this is not to discourage the use of older or lower-level languages for SDF ⎯ the knowledge units below can be suitably adapted for the actual language used.

The emergence of generative AI / LLMs, which can generate programs for many programming tasks, will undoubtedly affect the programming profession and consequently the teaching of many CS topics. However, we feel that to be able to effectively use Generative AI in programming tasks, a programmer must have a good understanding of programs, and hence must still learn the foundations of programming and develop basic programming skills - which is the aim of SDF. Consequently, we feel that the desired outcomes for SDF should remain the same, though different instructors may now give more emphasis to program understanding, documenting, specifications, analysis, and testing. (This is similar to teaching students multiplication and tables, addition, etc. even though calculators can do all this).

Changes since CS 2013

The main change from 2013 is a stronger emphasis on developing fundamental programming skills and effective use of in-built data structures (which many contemporary languages provide) for problem solving.

Overview

This Knowledge Area has five Knowledge Units. These are:

1. **SDF-Fundamentals**: Fundamental Programming Concepts and Practices: This knowledge unit aims to develop understanding of basic concepts, and ability to fluently use basic language constructs as well as modularity constructs. It also aims to familiarize students with the concept of common libraries and frameworks, including those to facilitate API-based access to resources.
2. **SDF-DataStructures**: Fundamental Data Structures: This knowledge unit aims to develop core concepts relating to Data Structures and associated operations. Students should understand the important data structures available in the programming language or as libraries, and how to use them effectively, including choosing appropriate data structures while designing solutions for a given problem.
3. **SDF-Algorithms**: Algorithms: This knowledge unit aims to develop the foundations of algorithms and their analysis. The KU should also empower students in selecting suitable algorithms for building modest-complexity applications.
4. **SDF-Practices**: Software Development Practices: This knowledge unit develops the core concepts relating to modern software development practices. Its aim is to develop student understanding and basic competencies in program testing, enhancing readability of programs, and using modern methods and tools including some general-purpose IDE.
5. **SDF-SEP**: Society, Ethics and Professionalism: This knowledge unit aims to develop an initial understanding of some of the ethical issues related to programming, professional values programmers need to have, and the responsibility to society that programmers have. This knowledge unit is a part of the SEP Knowledge Area.

Core Hours

Knowledge Units	CS Core	KA Core
SDF-Fundamentals: Fundamental Programming Concepts and Practices	20	
SDF-DataStructures: Fundamental Data Structures	12	
SDF-Algorithms: Algorithms	6	
SDF-Practices: Software Development Practices	5	
Total	43	

Knowledge Units

SDF-Fundamentals: Fundamental Programming Concepts and Practices

CS Core:

1. Basic concepts such as variables, primitive data types, expressions and their evaluation.
2. How imperative programs work: state and state transitions on execution of statements, flow of control.
3. Basic constructs such as assignment statements, conditional and iterative statements, basic I/O (using console).
4. Key modularity constructs such as functions (and methods and classes, if supported in the language) and related concepts like parameter passing, scope, abstraction, data encapsulation.
5. Input and output using files and APIs.
6. Structured data types available in the chosen programming language like sequences (e.g., arrays, lists), associative containers (e.g., dictionaries, maps), others (e.g., sets, tuples) and when and how to use them. (See also: AL-Fundamentals)
7. Libraries and frameworks provided by the language (when/where applicable).
8. Recursion.
9. Dealing with runtime errors in programs (e.g., exception handling).
10. Basic concepts of programming errors, testing, and debugging.
11. Documenting/commenting code at the program and module level.

Illustrative Learning Outcomes

In these learning outcomes, the term "Develop" means "design, write, test and debug".

1. Develop programs that use the fundamental programming constructs: assignment and expressions, basic I/O, conditional and iterative statements.
2. Develop programs using functions with parameter passing.
3. Develop programs that effectively use the different structured data types provided in the language like arrays/lists, dictionaries, and sets.
4. Develop programs that use file I/O to provide data persistence across multiple executions.
5. Develop programs that use language-provided libraries and frameworks (where applicable).
6. Develop programs that use APIs to access or update data (e.g., from the web).
7. Develop programs that create simple classes and instantiate objects of those classes (if supported by the language).
8. Explain the concept of recursion, and identify when and how to use it effectively.
9. Develop recursive functions.
10. Develop programs that can handle runtime errors.
11. Read a given program and explain what it does.
12. Write comments for a program or a module specifying what it does.
13. Trace the flow of control during the execution of a program.
14. Use appropriate terminology to identify elements of a program (e.g., identifier, operator, operand).

SDF-ADT: Fundamental Data Structures

CS Core: **(**See also: AL-Fundamentals**)**

1. Standard abstract data types such as lists, stacks, queues, sets, and maps/dictionaries, and operations on them.
2. Selecting and using appropriate data structures.
3. Performance implications of choice of data structure(s).
4. Strings and string processing.

Illustrative Learning Outcomes

1. Write programs that use each of the key abstract data types / data structures provided in the language (e.g., arrays, tuples/records/structs, lists, stacks, queues, and associative data types like sets, dictionaries/maps).
2. Select the appropriate data structure for a given problem.
3. Explain how the performance of a program may change when using different data structures or operations. .
4. Write programs that work with text by using string processing capabilities provided by the language.

SDF-Algorithms: Algorithms

CS Core: **(**See also: AL-Fundamentals**)**

1. Concept of algorithm and notion of algorithm efficiency.
2. Some common algorithms (e.g., sorting, searching, tree traversal, graph traversal).
3. Impact of algorithms on time/space efficiency of programs.

Illustrative Learning Outcomes

1. Explain the role of algorithms for writing programs.
2. Demonstrate how a problem may be solved by different algorithms, each with different properties.
3. Explain some common algorithms (eg., sorting, searching, tree traversal, graph traversal).
4. Explain the impact on space/time performance of some algorithms.

SDF-Practices: Software Development Practices

CS Core: (See also: SE-Construction)

1. Basic testing including test case design.
2. Use of a general-purpose IDE, including its debugger.
3. Programming style that improves readability.
4. Specifying functionality of a module in a natural language.

Illustrative Learning Outcomes

1. Develop tests for modules, and apply a variety of strategies to design test cases.
2. Explain some limitations of testing programs.
3. Build, execute and debug programs using a modern IDE and associated tools such as visual debuggers.
4. Apply basic programming style guidelines to aid readability of programs such as comments, indentation, proper naming of variables, etc.
5. Write specifications of a module as module comment describing its functionality.

SDF-SEP: Society, Ethics and Professionalism

CS Core:

1. Intellectual property rights of programmers for programs they develop
2. Plagiarism & academic integrity
3. Responsibility and liability of programmers regarding code they develop for solutions
4. Basic professional work ethics of programmers

Illustrative Learning Outcomes

1. Explain/understand some of the intellectual property issues relating to programs
2. Explain/understand when code developed by others can be used and proper ways of disclosing their use
3. Explain/understand the responsibility of programmers when developing code for an overall solution (which may be developed by a team)
4. Explain/understand one or more codes of conduct applicable to programmers

Professional Dispositions

- Self-Directed. Seeking out solutions to issues on their own (e.g., using technical forums, FAQs, discussions).
- Experimental. Practical experimentation characterized by experimenting with language features to understand them, quickly prototyping approaches, and using the debugger to understand why a bug is occurring.
- Technical curiosity. Characterized by, for example, interest in understanding how programs are executed, how programs and data are stored in memory.
- Technical adaptability. Characterized by willingness to learn and use different tools and technologies that facilitate software development.
- Perseverance. To continue efforts until, for example, a bug is identified, a program is robust and handles all situations, etc.
- Systematic. Characterized by attention to detail and use of orderly processes in practice.

Math Requirements

As SDF focuses on the first year and is foundational, it assumes only basic math knowledge that students acquire in school.

Shared Topics and Crosscutting Themes

Shared Topics:

- Topics 1, 2, 3: SDF-Algorithms: Algorithms :: AL-A
- Topics 1, 2, 3: SDF-Practices: Software Development Practices :: SE-Construction:

Course Packaging Suggestions

The SDF KA will generally be covered in introductory courses, often called CS1 and CS2. How much of the SDF KA can be covered in CS1 and how much is to be left for CS2 is likely to depend on the choice of programming language for CS1. For languages like Python or Java, CS1 can cover all the Programming Concepts and Development Methods KAs, and some of the Data Structures KA. It is desirable that they be further strengthened in CS2. The topics under algorithms KA and some topics under data structures KA can be covered in CS2. In case CS1 uses a language with fewer in-built data structures, then much of the Data Structures KA and some aspects of the programming KA may also need to be covered in CS2. With the former approach, the introductory course in programming can include the following:

1. SDF-Fundamentals: Fundamental Programming Concepts and Practices, 20 hours
2. SDF-DataStructures: Fundamental Data Structures, 12 hours
3. SDF-Algorithms: Algorithms, 6 hours
4. SDF-Practices: Software Development Practices, 5 hours
5. SDF-SEP: Society, Ethics and Professionalism

Pre-requisites: School Mathematics (Sets, Relations, Functions, and Logic)

Skill statement: A student who completes this course should be able to:

- Design, code, test, and debug a modest sized program that effectively uses functional abstraction.
- Select and use the appropriate language provided data structure for a given problem (like: arrays, tuples/records/structs, lists, stacks, queues, and associative data types like sets, dictionaries/maps.)

- Design, code, test, and debug a modest-sized object-oriented program using classes and objects.
- Design, code, test, and debug a modest-sized program that uses language provided libraries and frameworks (including accessing data from the web through APIs).
- Read and explain given code including tracing the flow of control during execution.
- Write specifications of a program or a module in natural language explaining what it does.
- Build, execute and debug programs using a modern IDE and associated tools such as visual debuggers.
- Explain the key concepts relating to programming like parameter passing, recursion, runtime exceptions and exception handling.

Committee

Chair: Pankaj Jalote, Chair, IIIT-Delhi, Delhi, India

Members:

- Brett A. Becker, University College Dublin, Dublin, Ireland
- Titus Winters, Google, New York City, NY, USA
- Andrew Luxton-Reilly, University of Auckland, Auckland, New Zealand
- Christian Servin, El Paso Community College, El Paso, TX, USA
- Karen Reid, University of Toronto, Toronto, Canada
- Adrienne Decker, University at Buffalo, Buffalo, NY, USA

Software Engineering (SE)

Preamble

As far back as the early 1970s, Brian Randell allegedly said, “Software engineering is the multi-person construction of multi-version programs.” This is an essential insight: while programming is the skill that governs our ability to write a program, software engineering is distinct in two dimensions: time and people.

First, a software engineering project is a team endeavor. Being a solitary programming expert is insufficient. Skilled software engineers will additionally demonstrate expertise in communication and collaboration. Programming may be an individual activity, but software engineering is a collaborative one, deeply tied to issues of professionalism, teamwork, and communication.

Second, a software engineering project is usually “multi-version.” It has an expected lifespan; it needs to function properly for months, years, or decades. Features may be added or removed to meet product requirements. The technological context will change, as our computing platforms evolve, programming languages change, dependencies upgrade, etc. This exposure to matters of time and change is novel when compared to a programming project: it isn’t enough to build a thing that works, instead it must work and stay working. Many of the most challenging topics in tech share “time will lead to change” as a root cause: backward compatibility, version skew, dependency management, schema changes, protocol evolution.

Software engineering presents a particularly difficult challenge for learning in an academic setting. Given that the major differences between programming and Software engineering are time and teamwork, it is hard to generate lessons that *require* successful teamwork and that faithfully present the risks of time. Additionally, some topics in software engineering will be more authentic and more relevant if and when our learners experience collaborative and long-term software engineering projects *in vivo* rather than in the classroom. Regardless of whether that happens as an internship, involvement in an open source project, or full-time engineering role, a month of full-time hands-on experience has more available hours than the average software engineering course.

Thus, a software engineering curriculum should focus primarily on ideas that are needed by a majority of new-grad hires, and that either are novel for those who are trained primarily as programmers, or that are abstract concepts that may not get explicitly stated/shared on the job. Such topics include, but are not limited to:

- Testing
- Teamwork, collaboration
- Communication
- Design
- Maintenance and Evolution
- Software engineering tools

Some such material is reasonably suited to a standard lecture or lecture+lab course. Discussing theoretical underpinnings of version control systems, or branching strategies in such systems, can be an effective way to familiarize students with those ideas. Similarly, a theoretical discussion can highlight the difference between static and dynamic analysis tools, or may motivate discussion of diamond dependency problems in dependency networks.

On the other hand, many of the fundamental topics of software engineering are best experienced in a hands-on fashion. Historically, project-oriented courses have been a common vehicle for such learning. We believe that such experience is valuable but also bears some interesting risks: students may form erroneous notions about the difficulty / complexity of collaboration if their only exposure is a single project with teams formed of other novice software engineers. It falls to instructors to decide on the right balance between theoretical material and hands-on projects - neither is a perfect vehicle for this challenging material. We strongly encourage instructors of project courses to aim for iteration and fast feedback - a few simple tasks repeated (i.e., in an Agile-structured project) is better than singular high-friction introductions to many types of tasks. Programs with real-world industry partners and clients are also particularly encouraged. If long-running project courses are not an option, anything that can expose learners to the collaborative and long-term aspects of software engineering is valuable: adding features to an existing codebase, collaborating on distinct parts of a larger whole, pairing up to write an encoder and decoder, etc.

All evidence suggests that the role of software in our society will continue to grow for the foreseeable future, and yet the era of “two programmers in a garage” seems to have drawn to a close. Most important software these days is clearly a team effort, building on existing code and leveraging existing functionality. The study of software engineering skills is a deeply important counterpoint to the everyday experience of computing students - we *must* impress on them the reality that few software projects are managed by writing from scratch as a solo endeavor. Communication, teamwork, planning, testing, and tooling are far more important as our students move on from the classroom and make their mark on the wider world.

Changes since CS 2013

This document shifts the focus of the Software Engineering knowledge area in a few ways compared to the goals of CS2013. The common reasoning behind most of these changes is to focus on material that learners would not pick up elsewhere in the curriculum, and that will be relevant *immediately* upon graduation, rather than at some future point in their careers.

- More explicit focus on the software workflow (version control, testing, code review, tooling)
- Less focus on team *leadership* and project management.
- More focus on team *participation*, communication, and collaboration

Overview

SE-Teamwork: Because of the nature of learning programming, most students in introductory SE have little or no exposure to the collaborative nature of SE. Practice (for instance in project work) may help, but lecture and discussion time spent on the value of clear, effective, and efficient communication and collaboration. are essential for Software Engineering.

SE-Tools: Industry reliance on SE tools has exploded in the past generation, with version control becoming ubiquitous, testing frameworks growing in popularity, increased reliance on static and dynamic analysis in practice, and near-ubiquitous use of continuous integration systems. Increasingly powerful IDEs provide code searching and indexing capabilities, as well as small scale refactoring tools and integration with other SE tools. An understanding of the nature of these tools is broadly valuable - especially version control systems.
SE-Requirements: Knowing how to build something is of little help if we do not know what to build. Product Requirements (aka Requirements Engineering, Product Design, Product Requirements solicitation, PRDs, etc.) introduces students to the processes surrounding the specification of the broad requirements governing development of a new product or feature.
SE-Design: While Product Requirements focuses on the user-facing functionality of a software system, Software Design focuses on the engineer-facing design of internal software components. This encompasses large design concerns such as software architecture, as well as small-scale design choices like API design.
SE-Construction: Software Construction focuses on practices that influence the direct production of software: use of tests, test driven development, coding style. More advanced topics extend into secure coding, dependency injection, work prioritization, etc.
SE-Validation: Software Verification and Validation focuses on how to improve the value of testing - understand the role of testing, failure modes, and differences between good tests and poor ones.
SE-Refactoring: Refactoring and Code Evolution focuses on refactoring and maintenance strategies, incorporating code health, use of tools, and backwards compatibility considerations.
SE-Reliability: Software Reliability aims to improve understanding of and attention to error cases, failure modes, redundancy, and reasoning about fault tolerance.
SE-FormalMethods: Formal Methods provides mathematically rigorous mechanisms to apply to software, from specification to verification. (Prerequisites: Substantial dependence on core material from the Discrete Structures area, particularly knowledge units DS/Basic Logic and DS/Proof Techniques.)

Core Hours

Knowledge Units	**CS Core**	**KA Core**
Teamwork	2	2
Tools and Environments	1	3
Product Requirements		2
Software Design	1	4
Software Construction	1	3
Software Verification and Validation	1	3
Refactoring and Code Evolution		2
Software Reliability		2
Formal Methods		
Total	**6**	**21**

Knowledge Units

SE-A: Teamwork (SE-Teamwork)

CS Core:

1. Effective communication, including oral and written, as well as formal (email, docs, comments, presentations) and informal (team chat, meetings) (See also: SEP/Professional Communication)
2. Common causes of team conflict, and approaches for conflict resolution
3. Cooperative programming
    a. Pair programming or Swarming
    b. Code review
    c. Collaboration through version control
4. Roles and responsibilities in a software team (See also: SEP/Professional Ethics)
    a. Advantages of teamwork
    b. Risks and complexity of such collaboration
5. Team processes
    a. Responsibilities for tasks, effort estimation, meeting structure, work schedule
6. Importance of team diversity and inclusivity (See also: SEP/Professional Communication)

KA Core:

7. Interfacing with stakeholders, as a team
    a. Management & other non-technical teams
    b. Customers
    c. Users
8. Risks associated with physical, distributed, hybrid and virtual teams
    a. Including communication, perception, structure, points of failure, mitigation and recovery, etc.

Illustrative Learning Outcomes:
CS Core:

1. Follow effective team communication practices.
2. Articulate the sources of, hazards of, and potential benefits of team conflict - especially focusing on the value of disagreeing about ideas or proposals without insulting people.
3. Facilitate a conflict resolution and problem solving strategy in a team setting.
4. Collaborate effectively in cooperative development/programming.
5. Propose and delegate necessary roles and responsibilities in a software development team.
6. Compose and follow an agenda for a team meeting.
7. Facilitate through involvement in a team project, the central elements of team building, establishing healthy team culture, and team management including creating and executing a team work plan.
8. Promote the importance of and benefits that diversity and inclusivity brings to a software development team

KA Core:

9. Reference the importance of, and strategies to, as a team, interface with stakeholders outside the team on both technical and non-technical levels.
10. Enumerate the risks associated with physical, distributed, hybrid and virtual teams and possible points of failure and how to mitigate against and recover/learn from failures.

SE-B: Tools and Environments (SE-Tools)

CS Core:

1. Software configuration management and version control (See also: SDF/Software Development)
   a. Configuration in version control, reproducible builds/configuration
   b. Version control branching strategies. Development branches vs. release branches. Trunk-based development.
   c. Merging/rebasing strategies, when relevant.

KA Core:

2. Release management
3. Testing tools including static and dynamic analysis tools (See also: SDF/Software Development)
4. Software process automation
   a. Build systems - the value of fast, hermetic, reproducible builds, compare/contrast approaches to building a project
   b. Continuous Integration (CI) - the use of automation and automated tests to do preliminary validation that the current head/trunk revision builds and passes (basic) tests
   c. Dependency management - updating external/upstream dependencies, package management, SemVer
5. Design and communication tools (docs, diagrams, common forms of design diagrams like UML)
6. Tool integration concepts and mechanisms (See also: SDF/Software Development)
7. Use of modern IDE facilities - debugging, refactoring, searching/indexing, ML-powered code assistants, etc. (See also: SDF/Software Development)

Illustrative Learning Outcomes:
CS Core:

1. Describe the difference between centralized and distributed software configuration management.
2. Describe how version control can be used to help manage software release management.
3. Identify configuration items and use a source code control tool in a small team-based project.

KA Core:

4. Describe how available static and dynamic test tools can be integrated into the software development environment.
5. Understand the use of CI systems as a ground-truth for the state of the team's shared code (build and test success).
6. 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.
7. Demonstrate the capability to use software tools in support of the development of a software product of medium size.

SE-C: Product Requirements (SE-Requirements)

KA Core:

1. Describe functional requirements using, for example, use cases or user stories
    a. Using at least one method of documenting and structuring functional requirements
    b. Understanding how the method supports design and implementation
    c. Strengths and weaknesses of using a particular approach
2. Properties of requirements including consistency, validity, completeness, and feasibility
3. Requirements elicitation
    a. Sources of requirements, for example, users, administrators, or support personnel
    b. Methods of requirement gathering, for example, surveys, interviews, or behavioral analysis
4. Non-functional requirements, for example, security, usability, or performance (aka Quality Attributes)
5. Risk identification and management
6. Communicating and/or formalizing requirement specifications

Non-core:

7. Prototyping
    a. A tool for both eliciting and validating/confirming requirements
8. Product evolution
    a. When requirements change, how to understand what effect that has and what changes need to be made
9. Effort estimation
    a. Learning techniques for better estimating the effort required to complete a task
    b. Practicing estimation and comparing to how long tasks actually take
    c. Effort estimation is quite difficult, so students are likely to be way off in many cases, but seeing the process play out with their own work is valuable

Illustrative Learning Outcomes:
KA Core:

1. Compare different methods of eliciting requirements along multiple axes.
2. Identify differences between two methods of describing functional requirements (e.g., customer interviews, user studies, etc.) and the situations where each would be preferred.
3. Identify which behaviors are required, allowed, or barred from a given set of requirements and a list of candidate behaviors.
4. Collect a set of requirements for a simple software system.
5. Identify areas of a software system that need to be changed, given a description of the system and a set of new requirements to be implemented.
6. Identify the functional and non-functional requirements in a set of requirements.

Non-core:

7. Create a prototype of a software system to validate a set of requirements. (Building a mock-up, MVP, etc.)
8. Estimate the time to complete a set of tasks, then compare estimates to the actual time taken.
9. Determine an implementation sequence for a set of tasks, adhering to dependencies between them, with a goal to retire risk as early as possible.
10. Write a requirement specification for a simple software system.

SE-D: Software Design (SE-Design)

CS Core:

1. System design principles (See also: SF/System Reliability)
    a. Levels of abstraction (e.g., architectural design and detailed design)
    b. Separation of concerns
    c. Information hiding
    d. Coupling and cohesion
2. Software architecture (See also: SF/System Reliability)
    a. Design paradigms
        i. Top-down functional decomposition / layered design
        ii. Data-oriented architecture
        iii. Object-oriented analysis and design
        iv. Event-driven design
    b. Standard architectures (e.g., client-server and microservice architectures including REST discussions, n-layer, pipes-and-filters, Model View Controller)
    c. Identifying component boundaries and dependencies
3. Programming in the large vs. programming in the small (See also: SF/System Reliability)
4. Code smells and other indications of code quality, distinct from correctness.

KA Core:

5. API design principles
    a. Consistency
        i. Consistent APIs are easier to learn and less error-prone
        ii. Consistency is both internal (between different portions of the API) and external (following common API patterns)
    b. Composability
    c. Documenting contracts
        i. API operations should describe their effect on the system, but not generally their implementation
        ii. Preconditions, postconditions, and invariants
    d. Expandability
    e. Error reporting
        i. Errors should be clear, predictable, and actionable
        ii. Input that does not match the contract should produce an error
        iii. Errors that can be reliably managed without reporting should be managed
6. Identifying and codifying data invariants and time invariants
7. Structural and behavioral models of software designs
8. Data design (See also: IM/Data Modeling)
    a. Data structures
    b. Storage systems

9. Requirement traceability
    a. Understanding which requirements are satisfied by a design

Non-Core:

10. Design modeling, for instance with class diagrams, entity relationship diagrams, or sequence diagrams
11. Measurement and analysis of design quality
12. Principles of secure design and coding (See also: SEC/Security Analysis and Engineering)
    a. Principle of least privilege
    b. Principle of fail-safe defaults
    c. Principle of psychological acceptability
13. Evaluating design tradeoffs (e.g., efficiency vs. reliability, security vs. usability)

Illustrative Learning Outcomes:

CS Core:

1. Identify the standard software architecture of a given high-level design.
2. Select and use an appropriate design paradigm to design a simple software system and explain how system design principles have been applied in this design.
3. Adapt a flawed system design to better follow principles such as separation of concerns or information hiding.
4. Identify the dependencies among a set of software components in an architectural design.

KA Core:

5. Design an API for a single component of a large software system, including identifying and documenting each operation's invariants, contract, and error conditions.
6. Evaluate an API description in terms of consistency, composability, and expandability.
7. Expand an existing design to include a new piece of functionality.
8. Design a set of data structures to implement a provided API surface.
9. Identify which requirements are satisfied by a provided software design.

Non-Core:

10. Translate a natural language software design into class diagrams.
11. Adapt a flawed system design to better follow the principles of least privilege and fail-safe defaults.
12. Contrast two software designs across different qualities, such as efficiency or usability.

SE-E: Software Construction (SE-Construction)

CS Core:

1. Practical small-scale testing (See also: SDF/Software Development Practices)
    a. Unit testing
    b. Test-driven development - This is particularly valuable for students psychologically, as it is far easier to engage constructively with the challenge of identifying challenging inputs for a given API (edge cases, corner cases) a priori. If they implement first, the instinct is often to avoid trying to crash their new creation, while a test-first approach gives them the intellectual satisfaction of spotting the problem cases and then watching as more tests pass during the development process.
2. Documentation (See also: SDF/Software Development Practices)
    a. Interface documentation - Describe interface requirements, potentially including (formal or informal) contracts, pre and post conditions, invariants.

    b. Implementation documentation should focus on tricky and non-obvious pieces of code, whether because the code is using advanced language features or the behavior of the code is complex. (Do not add comments that re-state common/obvious operations and simple language features.)
        i. Clarify dataflow, computation, etc., focusing on what the code is
        ii. Identify subtle/tricky pieces of code and refactor to be self-explanatory if possible, or provide appropriate comments to clarify.

KA Core:

3. Coding style (See also: SDF/Software Development Practices)
    a. Style guides
    b. Commenting
    c. Naming
4. "Best Practices" for coding: techniques, idioms/patterns, mechanisms for building quality programs (See also: SEC/Defensive Programming, SDF/Software Development Practices)
    a. Defensive coding practices
    b. Secure coding practices and principles
    c. Using exception handling mechanisms to make programs more robust, fault-tolerant
5. Debugging (See also: SDF/Software Development Practices)
6. Logging
7. Use of libraries and frameworks developed by others (See also: SDF/Software Development Practices)

Non-Core:

8. Larger-scale testing
    a. Test doubles (stubs, mocks, fakes)
    b. Dependency injection
9. Work sequencing, including dependency identification, milestones, and risk retirement
    a. Dependency identification: Identifying the dependencies between different tasks
    b. Milestones: A collection of tasks that serve as a marker of progress when completed. Ideally, the milestone encompasses a useful unit of functionality.
    c. Risk retirement: Identifying what elements of a project are risky and prioritizing completing tasks that address those risks
10. Potential security problems in programs (See also: SEC/Defensive Programming)
    a. Buffer and other types of overflows
    b. Race conditions
    c. Improper initialization, including choice of privileges
    d. Input validation
11. Documentation (autogenerated)
12. Development context: "green field" vs. existing code base
    a. Change impact analysis
    b. Change actualization
13. Release management
14. DevOps practices

Illustrative Learning Outcomes:

CS Core:

1. Write appropriate unit tests for a small component (several functions, a single type, etc.).
2. Write appropriate interface and (if needed) implementation comments for a small component.

KA Core:

3. Describe techniques, coding idioms and mechanisms for implementing designs to achieve desired properties such as reliability, efficiency, and robustness.
4. Write robust code using exception handling mechanisms.
5. Describe secure coding and defensive coding practices.
6. Select and use a defined coding standard in a small software project.
7. Compare and contrast integration strategies including top-down, bottom-up, and sandwich integration.
8. Describe the process of analyzing and implementing changes to code base developed for a specific project.
9. Describe the process of analyzing and implementing changes to a large existing code base.

Non-Core:

10. Rewrite a simple program to remove common vulnerabilities, such as buffer overflows, integer overflows and race conditions.
11. Write a software component that performs some non-trivial task and is resilient to input and run-time errors.

SE-F: Software Verification and Validation (SE-Validation)

CS Core:

1. Verification and validation concepts
    a. Verification: Are we building the thing right?
    b. Validation: Did we build the right thing?
2. Why testing matters
    a. Does the component remain functional as the code evolves?
3. Testing objectives
    a. Usability
    b. Reliability
    c. Conformance to specification
    d. Performance
    e. Security
4. Test kinds
    a. Unit
    b. Integration
    c. Validation
    d. System
5. Stylistic differences between tests and production code
    a. DAMP vs. DRY - more duplication is warranted in test code.

KA Core:

6. Test planning and generation
    a. Test case generation, from formal models, specifications, etc.
    b. Test coverage
        i. Test matrices
        ii. Code coverage (how much of the code is tested)
        iii. Environment coverage (how many hardware architectures, OSes, browsers, etc. are tested)
    c. Test data and inputs
7. Test development
    a. Test-driven development
    b. Object oriented testing, mocking, and dependency injection

   c. Black-box and white-box testing techniques
   d. Test tooling, including code coverage, static analysis, and fuzzing
8. Verification and validation in the development cycle
   a. Code reviews
   b. Test automation, including automation of tooling
   c. Pre-commit and post-commit testing
   d. Trade-offs between test coverage and throughput/latency of testing
   e. Defect tracking and prioritization
      i. Reproducibility of reported defects
9. Domain specific verification and validation challenges
   a. Performance testing and benchmarking
   b. Asynchrony, parallelism, and concurrency
   c. Safety-critical
   d. Numeric

Non-Core:

10. Verification and validation tooling and automation
   a. Static analysis
   b. Code coverage
   c. Fuzzing
   d. Dynamic analysis and fault containment (sanitizers, etc.)
   e. Fault logging and fault tracking
11. Test planning and generation
   a. Fault estimation and testing termination including defect seeding
   b. Use of random and pseudo random numbers in testing
12. Performance testing and benchmarking
   a. Throughput and latency
   b. Degradation under load (stress testing, FIFO vs. LIFO handling of requests)
   c. Speedup and scaling
      i. Amadhl's law
      ii. Gustafson's law
      iii. Soft and weak scaling
   d. Identifying and measuring figures of merits
   e. Common performance bottlenecks
      i. Compute-bound
      ii. Memory-bandwidth bound
      iii. Latency-bound
   f. Statistical methods and best practices for benchmarking
      i. Estimation of uncertainty
      ii. Confidence intervals
   g. Analysis and presentation (graphs, etc.)
   h. Timing techniques
13. Testing asynchronous, parallel, and concurrent systems
14. Verification and validation of non-code artifacts (documentation, training materials)

Illustrative Learning Outcomes:

CS Core:

1. Explain why testing is important.
2. Distinguish between program validation and verification.
3. Describe different objectives of testing.
4. Compare and contrast the different types and levels of testing (regression, unit, integration, systems, and acceptance).

KA Core:

5. Describe techniques for creating a test plan and generating test cases.
6. Create a test plan for a medium-size code segment which includes a test matrix and generation of test data and inputs.
7. Implement a test plan for a medium-size code segment.
8. Identify the fundamental principles of test-driven development methods and explain the role of automated testing in these methods.
9. Discuss issues involving the testing of object-oriented software.
10. Describe mocking and dependency injection and their application.
11. Undertake, as part of a team activity, a code review of a medium-size code segment.
12. Describe the role that tools can play in the validation of software.
13. Automate testing in a small software project.
14. Explain the roles, pros, and cons of pre-commit and post-commit testing.
15. Discuss the tradeoffs between test coverage and test throughput/latency and how this can impact verification.
16. Use a defect tracking tool to manage software defects in a small software project.
17. Discuss the limitations of testing in certain domains.

Non-Core:

18. Describe and compare different tools for verification and validation.
19. Automate the use of different tools in a small software project.
20. Explain how and when random numbers should be used in testing.
21. Describe approaches for fault estimation.
22. Estimate the number of faults in a small software application based on fault density and fault seeding.
23. Describe throughput and latency and provide examples of each.
24. Explain speedup and the different forms of scaling and how they are computed.
25. Describe common performance bottlenecks.
26. Describe statistical methods and best practices for benchmarking software.
27. Explain techniques for and challenges with measuring time when constructing a benchmark.
28. Identify the figures of merit, construct and run a benchmark, and statistically analyze and visualize the results for a small software project.
29. Describe techniques and issues with testing asynchronous, concurrent, and parallel software.
30. Create a test plan for a medium-size code segment which contains asynchronous, concurrent, and/or parallel code, including a test matrix and generation of test data and inputs.
31. Describe techniques for the verification and validation of non-code artifacts.

SE-G: Refactoring and Code Evolution (SE-Refactoring)

KA Core:

1. Hyrum's Law / The Law of Implicit Interfaces
2. Backward compatibility
   a. Compatibility is not a property of a single entity, it's a property of a *relationship*.
   b. Backward compatibility needs to be evaluated in terms of provider + consumer(s) or with a well-specified model of what forms of compatibility a provider aspires to / promises.
3. Refactoring
   a. Standard refactoring patterns (rename, inline, outline, etc.)
   b. Use of refactoring tools in IDE
   c. Application of static-analysis tools (to identify code in need of refactoring, generate changes, etc.)

   d. Value of refactoring as a remedy for technical debt
4. Versioning
   a. Semantic Versioning (SemVer)
   b. Trunk-based development

Non-Core:

5. "Large Scale" Refactoring - techniques when a refactoring change is too large to commit safely (large projects), or when it is impossible to synchronize change between provider + all consumers (multiple repositories, consumers with private code).
   a. Express both old and new APIs so that they can co-exist
   b. Minimize the size of *behavior* changes
   c. Why these techniques are required, (e.g., "API consumers I can see" vs "consumers I can't see")

Illustrative Learning Outcomes:

KA-Core:

1. Identify both explicit and implicit behavior of an interface, and identify potential risks from Hyrum's Law
2. Consider inputs from static analysis tools and/or Software Design principles to identify code in need of refactoring.
3. Identify changes that can be broadly considered "backward compatible," potentially with explicit statements about what usage is or is not supported
4. Refactor the implementation of an interface to improve design, clarity, etc. with minimal/zero impact on existing users
5. Evaluate whether a proposed change is sufficiently safe given the versioning methodology in use for a given project

Non-Core:

6. Plan a complex multi-step refactoring to change default behavior of an API safely.

SE-H: Software Reliability (SE-Reliability)

KA Core:

1. Concept of reliability as probability of failure or mean time between failures, and faults as cause of failures
2. Identifying reliability requirements for different kinds of software
3. Software failures caused by defects/bugs, and so for high reliability goal is to have minimum defects - by injecting fewer defects (better training, education, planning), and by removing most of the injected defects (testing, code review, etc.)
4. Software reliability, system reliability and failure behavior
5. Defect injection and removal cycle, and different approaches for defect removal
6. Compare the "error budget" approach to reliability with the "error-free" approach, and identify domains where each is relevant

Non-Core:

7. Software reliability models
8. Software fault tolerance techniques and models
   a. Contextual differences in fault tolerance (e.g., crashing a flight critical system is strongly avoided, crashing a data processing system before corrupt data is written to storage is highly valuable)
9. Software reliability engineering practices - including reviews, testing, practical model checking

10. Identification of dependent and independent failure domains, and their impact on system reliability
11. Measurement-based analysis of software reliability - telemetry, monitoring and alerting, dashboards, release qualification metrics, etc.

Illustrative Learning Outcomes:
KA Core:
1. Describe how to determine the level of reliability required by a software system.
2. Explain the problems that exist in achieving very high levels of reliability.
3. Understand approaches to minimizing faults that can be applied at each stage of the software lifecycle.

Non-Core:
4. Demonstrate the ability to apply multiple methods to develop reliability estimates for a software system.
5. Identify methods that will lead to the realization of a software architecture that achieves a specified level of reliability.
6. Identify ways to apply redundancy to achieve fault tolerance.
7. Identify single-point-of-failure (SPF) dependencies in a system design.

SE-I: Formal Methods (SE-FormalMethods)

Non-Core:
1. Formal specification of interfaces
    a. Specification of pre- and post- conditions
    b. Formal languages for writing and analyzing pre- and post-conditions.
2. Problem areas well served by formal methods
    a. Lock-free programming, data races
    b. Asynchronous and distributed systems, deadlock, livelock, etc.
3. Comparison to other tools and techniques for defect detection
    a. Testing
    b. Fuzzing
4. Formal approaches to software modeling and analysis
    a. Model checkers
    b. Model finders

Illustrative 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.
2. Apply formal specification and analysis techniques to software designs and programs with low complexity.
3. Explain the potential benefits and drawbacks of using formal specification languages.

Professional Dispositions

- **Collaborative**: Software engineering is increasingly described as a “team sport” - successful software engineers are able to work with others effectively. Humility, respect, and trust underpin the collaborative relationships that are essential to success in this field.

- **Professional**: Software engineering produces technology that has the chance to influence literally billions of people. Awareness of our role in society, strong ethical behavior, and commitment to respectful day-to-day behavior outside of one's team are essential.
- **Communicative:** No single software engineer on a project is likely to know all of the project details. Successful software projects depend on engineers communicating clearly and regularly in order to coordinate effectively.
- **Meticulous:** Software engineering requires attention to detail and consistent behavior from everyone on the team. Success in this field is clearly influenced by a meticulous approach - comprehensive understanding, proper procedures, and a solid avoidance of cutting corners.
- **Accountable:** The collaborative aspects of software engineering also highlight the value of accountability. Failing to take responsibility, failing to follow through, and failing to keep others informed are all classic causes of team friction and bad project outcomes.

Math Requirements

Desirable:

- Introductory statistics (performance comparisons, evaluating experiments, interpreting survey results, etc.)

Course Packaging Suggestions

Advanced Course to include at least the following:

- SE-Teamwork: Teamwork - 4 hours
- SE-Tools: Tools and Environments - 4 hours
- SE-Requirements: Product Requirements - 2 hours
- SE-Design: Software Design - 5 hours
- SE-Construction: Software Construction - 4 hours
- SE-Validation: Verification and Validation - 4 hours
- SE-Refactoring: Refactoring and Code Evolution - 2 hours
- SE-Reliability: Software Reliability - 2 hours
- SEP-C: Professional Ethics and SEP-F: Professional Communication - 7 hours

Pre-requisites:

- SDF-A: Fundamental Programming Concepts and Practices

Skill statement: A student who completes this course should be able to perform good quality code review for colleagues (especially focusing on professional communication and teamwork needs), read and write unit tests, use basic software tools (IDEs, version control, static analysis tools) and perform basic activities expected of a new hire on a software team.

Committee

Chair: Titus Winters (Google, New York City, NY, USA)
Members:

- Brett A. Becker, University College Dublin, Dublin, Ireland
- Adam Vartanian, Cord, London, UK

- Bryce Adelstein Lelbach, NVIDIA, New York City, NY, USA
- Patrick Servello, CIWRO, Norman, OK, USA
- Pankaj Jalote, IIIT-Delhi, Delhi, India
- Christian Servin, El Paso Community College, El Paso, TX, USA

Contributors:

- Hyrum Wright, Google, Pittsburgh, PA, USA
- Olivier Giroux, Apple, Cupertino, CA, USA
- Gennadiy Civil, Google, New York City, NY, USA

Security (SEC)

Preamble

The world increasingly relies on computing infrastructure to support nearly every facet of modern critical infrastructure: transportation, communication, healthcare, education, energy generation and distribution, just to name a few. In recent years, with rampant attacks on and breaches of this critical computing infrastructure, it has become clearer that computer science graduates have an increased role in designing, implementing, and operating software systems that are secure and can keep information private.

In CS2023, the Security (SEC) Knowledge Area (KA) focuses on developing a *security mindset* into the overall ethos of computer science graduates so that security is inherent in all of their work products. The *Security* title choice was intentional to serve as a one-word umbrella term for this KA, which also includes concepts such as privacy, cryptography, secure system design, principles of modularity, and others that are imported from the other KAs. Reasons for this choice are discussed below; see also Figure 1.

The SEC KA also relies on shared concepts pervasive in all the other areas of CS2023. Additionally, the six cross-cutting themes of cybersecurity, as defined Cybersecurity Curricular 2017 (CSEC2017)[2], viewed with a computer science lens: confidentiality, integrity, availability, risk assessment, systems thinking, and adversarial thinking, are relevant here. In addition, the SEC KA adds a seventh cross-cutting theme: *human-centered thinking*, emphasizing that humans are also a link in the overall chain of security, a theme that needs to be inculcated into computer science students, along with *risk assessment* and *adversarial thinking,* which are not typically covered in other Computer Science Knowledge Areas (KAs). Students also need to learn security concepts such as authentication, authorization, and non-repudiation. They need to learn about system vulnerabilities and understand threats against software systems.

Principles of protecting systems (also in the Software Development Fundamentals and Software Engineering KAs) include security-by-design, privacy-by-design, and defense in depth. Another concept important in the SEC KA is the notion of assurance, which is an attestation that security mechanisms need to comply with the security policies that have been defined for data, processes, and systems. Assurance is tied in with the concepts verification and validation in the SE KA. With the increased use of computing systems and data sets in modern society, the issues of privacy, especially its technical aspects not covered in the Society, Ethics and Professionalism KA, become essential to computer science students.

[2] Joint Task Force on Cybersecurity Education. 2017. Cybersecurity Curricula 2017. ACM, IEEE-CS, AIS SIGSEC, and IFIP WG 11.8. https://doi.org/10.1145/3184594

Changes since CS 2013

The Security KA is an “updated” name for CS2013’s Information Assurance and Security (IAS) knowledge area. Since 2013, Information Assurance and Security has been rebranded as Cybersecurity, which has become a new computing discipline: the CSEC2017 curricular guidelines for this discipline have been developed by a Joint Task Force of the ACM, IEEE Computer Society, AIS and IFIP.

Moreover, since 2013, other curricular recommendations for cybersecurity beyond CS2013 and CSEC 2017 have been available. In the US, the Centers of Academic Excellence has Cyber Defense and Cyber Operations designations for institutions with cybersecurity programs that meet the CAE curriculum requirements, which are highly granular. Additionally, the US National Institute for Standards and Technologies (NIST) has developed and revised the National Initiative for Cybersecurity Education (NICE) Workforce Framework for Cybersecurity (NICE Framework), which identifies competencies (knowledge, and skills) needed to perform cybersecurity work. The European Cybersecurity Skills Framework (ECSF) includes a standard ontology to describe cybersecurity tasks, role and to address the cybersecurity shortage in EU member countries, and types. The computer science aspects of these guidelines also informed the content of this draft of the SEC KA.

Building on CS2013’s recognition of the pervasiveness of security in computer science, the CS2023 SEC KA focuses on ensuring that students develop the *security mindset* so that they are prepared for the continual changes occurring in computing. One noteworthy addition is the knowledge unit for security analysis and engineering to support the concepts of security-by-design and privacy-by-design.

Differences between the CS2023 Security KA and Cybersecurity

Feedback to earlier drafts of the SEC KA showed the need to clarify the differences between CS2023 SEC KA and the young computing-based discipline of cybersecurity. CS2023’s SEC KA, which is informed by the notion of a computer science disciplinary lens mentioned in CSEC 2017, focuses on those aspects of security, privacy, and related concepts important for computer science students. It builds primarily on security concepts already included in other CS2023 KAs. In short, the major goal of the SEC KA is to ensure computer science graduates to design and develop more secure code, ensure data security and privacy, and can apply the security mindset to their daily activities.

Protecting what happens *within* the perimeter is a core competency of computer science graduates. Although the computer science and cybersecurity knowledge units have overlaps, the demands upon cybersecurity graduates typically are to protect the perimeter. Cybersecurity is a highly interdisciplinary field of study that covers eight knowledge areas (data, software, component, connection, system, human, organizational, and societal security) and prepares its students for both technical and managerial positions. The first five knowledge areas are technical and have overlaps with the CS2023 SEC KA, but the intent of coverage is substantially different.

For instance, consider the data security knowledge unit. The computer science student will need to view this knowledge unit using the lens of computer science, as an extension of the material covered in CS2023’s Data Management KA while the cybersecurity student will need to view data security in the overall context of cybersecurity goals. These viewpoints are not totally distinct and have overlaps, but the lenses used to examine and present the content are different, as shown in Figure 1. x1Similar diagrams apply to the CS2023 SEC KAs overlaps with the CSEC 2017 KAs.

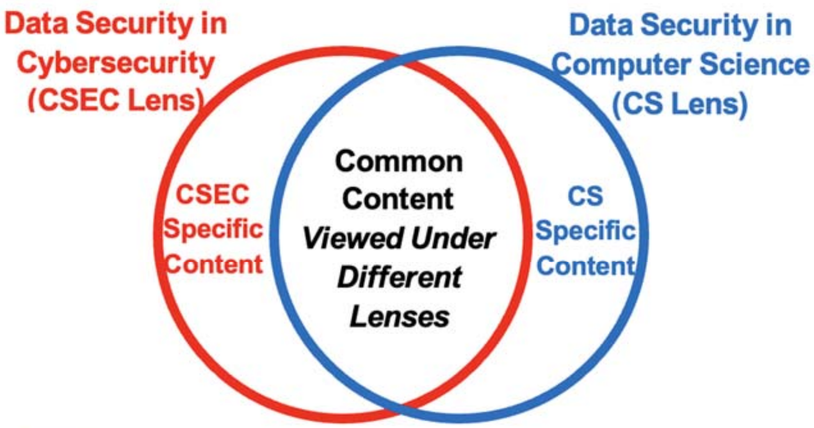

Data Security in Cybersecurity (CSEC Lens)
Data Security in Computer Science (CS Lens)
CSEC Specific Content
Common Content Viewed Under Different Lenses
CS Specific Content

Figure 1. Data Security – Cybersecurity versus CS2023 SEC.
(Other knowledge areas will have similar Venn diagrams)

Core Hours

Knowledge Unit	**CS Core**	**KA Core**
Foundational Security	2	6
Defensive Programming	2	5
Cryptography	1	4
Security Analysis and Engineering	1	9
Digital Forensics	0	6
Security Governance	0	3
Total	**6**	**33**

The SEC KA also relies on CS Core and KA Core hours from the other KAs, as discussed below in the Shared Concepts and Crosscutting Themes section. At least 28 hours of CS Core hours from the other KAs are needed, either to provide the basis for the SEC KA or to complement its content shown here.

Knowledge Units

SEC-Foundations: Foundational Security

CS Core:

1. Developing a security mindset, including crosscutting concepts: confidentiality, integrity, availability, risk assessment, systems thinking, adversarial thinking, human-centered thinking
2. Vulnerabilities, threats, and attack vectors
3. Denial of Service (DoS) and Distributed Denial of Service (DDoS)
4. Principles and practices of protection, e.g., least privilege, open design, fail-safe defaults, and defense in depth; and how they can be implemented
5. Principles and practices of privacy
6. Authentication and authorization
7. Tensions between security, privacy, performance, and other design goals
8. Applicability of laws and regulations on security and privacy
9. Ethical considerations for designing secure systems and maintaining privacy

KA Core:

10. Cryptographic building blocks, e.g., symmetric encryption, asymmetric encryption, hashing, and message authentication
11. Hardware considerations in security
12. Access control, e.g., discretionary, mandatory, role-based, and attribute-based
13. Intrusion detection systems
14. Principles of usable security and human-centered computing
15. Concepts of trust and trustworthiness
16. Applications of security mindset: web, cloud, and mobile devices.
17. Internet of Things (IoT) security and privacy
18. Newer access control approaches

Illustrative Learning Outcomes:

CS Core:

1. Evaluate a system for possible attacks that can be launched by any adversary
2. Design and develop approaches to protect a system from a set of identified threats
3. Design and develop a system designed to protect individual privacy

KA Core:

4. Evaluate a system for trustworthiness
5. Develop a system that incorporates various principles of security
6. Design and develop a web application ensuring data security and privacy
7. Evaluate a system for compliance to a given law
8. Show a system has been designed to avoid harm to user privacy

SEC-Defense: Defensive Programming

CS Core Topics

1. Common vulnerabilities and weaknesses
2. Input validation and data sanitization
3. Type safety and type-safe languages
4. Buffer overflows, stack smashing, and integer overflows
5. SQL injection and other injection attacks
6. Security issues due to race conditions

KA Core Topics

7. Using third-party components securely
8. Assurance: testing (including fuzzing), verification and validation
9. Static and dynamic analyses
10. Preventing information flow attacks
11. Offensive security: what, why. where, and how
12. Malware: varieties, creation, and defense against them
13. Ransomware and its prevention

Non-core Topics (including Emerging topics)

14. Secure compilers and secure code generation

Illustrative Learning Outcomes

CS Core

1. Explain the problems underlying in provided examples of an enumeration of common weaknesses, and how they can be circumvented
2. Explain the importance of defensive programming in showing compliance to various laws
3. Apply input validation and data sanitization techniques to enhance security of a program
4. Rewrite a program in a type-safe language (e.g., Java or Rust) originally written in an unsafe programming language, (e.g., C/C++)
5. Evaluate a program for possible buffer overflow attacks and rewrite to prevent such attacks
6. Evaluate a set of related programs for possible race conditions and prevent an adversary from exploiting them
7. Evaluate and prevent SQL injections attacks on a database application.

KA Core

8. Explain the risks with misusing interfaces with third-party code and how to correctly use third-party code
9. Discuss the need to update software to fix security vulnerabilities and the lifecycle management of the fix
10. List examples of information flows and prevent unauthorized flows

11. Demonstrate how programs are tested for input handling errors
12. Use static and dynamic tools to identify programming faults
13. Describe different kinds of malicious software and how to prevent them from occurring in a system
14. Explain what ransomware is and implement preventive techniques to reduce its occurrence

SEC-Cryptography

CS Core Topics

1. Mathematical preliminaries: modular arithmetic, Euclidean algorithm, probabilistic independence, linear algebra basics, number theory, finite fields, complexity, asymptotic analysis
2. Differences between algorithmic, applied, and math views of cryptography
3. History and real-world applications, e.g., electronic cash, secure channels between clients and servers, secure electronic mail, entity authentication, device pairing, voting systems
4. Classical cryptosystems, such as shift, substitution, transposition ciphers, code books, machines.
5. Basic cryptography: symmetric key and public key cryptography
6. Kerckhoff's principle and use of vetted libraries

KA Core Topics

7. Additional mathematical foundations: primality and factoring; elliptic curve cryptography
8. Private-key cryptosystems: substitution-permutation networks, linear cryptanalysis, differential cryptanalysis, DES, AES
9. Public-key cryptosystems: Diffie-Hellman, RSA
10. Data integrity and authentication: hashing, digital signatures
11. Cryptographic protocols: challenge-response authentication, zero-knowledge protocols, commitment, oblivious transfer, secure 2-party or multi-party computation, secret sharing, and applications
12. Attacker capabilities: chosen-message attack (for signatures), birthday attacks, side channel attacks, fault injection attacks.
13. Quantum cryptography
14. Blockchain and cryptocurrencies

Illustrative Learning Outcomes

CS Core

1. Describe the role of cryptography in supporting security and privacy
2. Describe the dangers of inventing one's own cryptographic methods
3. Describe the role of cryptography in supporting confidentially and privacy
4. Discuss the importance of prime numbers in cryptography and explain their use in cryptographic algorithms
5. Implement and cryptanalyze classical ciphers

KA Core

6. Describe modern private-key cryptosystems and ways to cryptanalyze them
7. Describe modern public-key cryptosystems and ways to cryptanalyze them
8. Compare different algorithms in their support for security
9. Explain key exchange protocols and show approaches to reduce their failure
10. Describe real-world applications of cryptographic primitives and protocols
11. Describe quantum cryptography and the impact of quantum computing on cryptographic algorithms

SEC-Engineering: Security Analysis and Engineering

CS Core Topics

1. Security engineering goals: building systems that remain dependable despite errors, accidents, or malicious adversaries
2. Problem analysis and situational analysis to address system security
3. Security Design, including security testing; security evaluation and assessment
4. Tradeoff analysis based on time, cost, risk tolerance, risk acceptance, return on investment, and so on

KA Core Topics

5. Security Analysis, covering security requirements analysis; security controls analysis; threat analysis; and vulnerability analysis
6. Security Attack Domains and Attack Surfaces, e.g., communications and Networking, hardware, physical, social engineering, software, and supply chain
7. Security Attack Modes, Techniques and Tactics, e.g., authentication abuse; brute force; buffer manipulation; code injection; content insertion; denial of service; eavesdropping; function bypass; impersonation; integrity attack; interception; phishing; protocol analysis; privilege abuse; spoofing; and traffic injection
8. Security Technical Controls: identity and credential subsystems; access control and authorization subsystems; information protection subsystems; monitoring and audit subsystems; integrity management subsystems; cryptographic subsystems

Illustrative Learning Outcomes

CS Core

1. Create a threat model for a system or system design
2. Apply situational analysis to develop secure solutions under a specified scenario
3. Evaluate a give scenario for tradeoff analysis for system performance, risk assessment, and costs

KA Core

4. Design a set of technical security controls, countermeasures and information protections to meet the security requirements and security objectives for a system

5. Develop a system that incorporates various principles of security
6. Evaluate the effectiveness of security functions, technical controls and componentry for a system.
7. Identify security vulnerabilities and weaknesses in a system
8. Mitigate threats, vulnerabilities and weaknesses in a system

SEC-Forensics: Digital Forensics

CS Core Topics

1. Not applicable

KA Core Topics

2. Basic principles and methodologies for digital forensics.
3. System design for forensics
4. Rules of evidence – general concepts and differences between jurisdictions
5. Legal issues: digital evidence protection and management, chains of custody, reporting, serving as an expert witness
6. Forensics in different situations: operating systems, file systems, application forensics, web forensics, network forensics, mobile device forensics, use of database auditing
7. Attacks on forensics and preventing such attacks

Illustrative Learning Outcomes

CS Core

1. Not applicable

KA Core

2. Explain what a digital investigation is and how it can be implemented
3. Design and implement software to support forensics
4. Describe legal requirements for using seized data and its usage
5. Describe and implement end-to-end chain of custody from initial digital evidence seizure to evidence disposition
6. Extract data from a hard drive to comply with the law
7. Describe a person's professional responsibility and liability when testifying as a forensics expert
8. Recover data based on a given search term from an imaged system
9. Reconstruct data and events from an application history, or a web artifact, or a cloud database, or a mobile device
10. Capture and interpret network traffic
11. Discuss the challenges associated with mobile device forensics
12. Apply forensics tools to investigate security breaches
13. Identify and mitigate anti-forensic methods

SEC-Governance: Security Governance

CS Core Topics

1. Not applicable

KA Core Topics

2. Protecting critical assets from threats
3. Security governance: organizational objectives and general risk assessment
4. Security management: achieve and maintain appropriate levels of confidentiality, integrity, availability, accountability, authenticity, and reliability
5. Approaches to identifying and mitigating risks to computing infrastructure:
6. Security controls: management, operational and technical controls
7. Policies for data collection, backups, and retention; cloud storage and services; breach disclosure

Illustrative Learning Outcomes

CS Core

1. Not applicable

KA Core

2. Describe critical assets and how they can be protected
3. Differentiate between security governance, management, and controls, giving examples of each
4. Describe a technical control and implement it
5. Identify and assess risk of programs and database applications causing breaches
6. Design and implement appropriate backups, given a policy
7. Discuss a breach disclosure policy based on legal requirements and implement the policy
8. Identify the risks and benefits of outsourcing to the cloud.

CS Core Topics

1. TBD
2.

KA Core Topics

3. TBD

Illustrative Learning Outcomes

CS Core

14. TBD

KA Core

15. TBD

Expand the following as knowledge units?

SEC-Additional KUs or else?

Adversarial Machine Learning

CyberAnalytics

Hardware and Security

Professional Dispositions

- **Meticulous:** students need to pay careful attention to details to ensure the protection of real-world software systems.
- **Self-directed**: students must be ready to deal with the many novel and easily unforeseeable ways in which adversaries might launch attacks.
- **Collaborative**: students must be ready to collaborate with others , as collective knowledge and skills will be needed to prevent attacks, protect systems and data during attacks, and plan for the future after the immediate attack has been mitigated.
- **Responsible:** students need to show responsibility when designing, developing, deploying, and maintaining secure systems, as their enterprise and society is constantly at risk.
- **Accountable**: students need to know that as future professionals that they will be held accountable if a system or data breach were to occur, which should strengthen their resolve to prevent such breaches from occurring in the first place.

Math Requirements

Required:

- Sets, Relations, Logical Operations, Number Theory, Prime factoring
- Linear Algebra: Arithmetic Operations, Matrix operations
- Basic probability and descriptive statistics

Desired:

- System performance evaluation: probability and experiment design.
- Elliptic curves

Course Packaging Suggestions

There are two suggestions for course packaging.

The first is to infuse the CS Core hours of the SEC KA into appropriate places in other coursework that covers related security topics in the following knowledge units, as mentioned in the Shared Concepts section above. It seems to reasonable to assume that as the CS Core Hours of the SEC KA are only 6 hours, one or more of the following KUs being covered could accommodate the additional hours.

- AL-E: Algorithms and Society
- AR-D: Memory Hierarchy
- AR-H: Heterogeneous Architectures
- DM-I: Data Security and Privacy
- FPL-H: Language Translation and Execution
- FPL-N: /Runtime Behavior and Systems
- FPL-G: Type Systems
- HCI/Human Factors and Security
- NC-F: Network Security
- OS-G: Protection and Safety
- PDC-B: Communication
- PDC-D: Software Engineering
- SDF-A: Fundamental Programming Concepts and Practices
- SDF-D: Software Development Practices
- SE-F: Software Verification and Validation
- SEP-E: Privacy and Civil Liberties
- SEP-J: Security Policies, Laws and Computer Crime
- SF-G: Systems Security
- SPD-A: Common Aspects/Shared Concerns
- SPD-C: Mobile Platforms
- SPD-B: Web Platforms

The second approach is to create an additional course that packages the following:

Introduction to Computer Security to include the following:

- SEC-A: Foundational Security – 8 hours
- SEC-B: Defensive Programming – 7 hour
- SEC-C: Cryptography – 5 hours
- SEC-D: Security Analysis and Engineering – 4 hours
- SEC-E: Digital Forensics – 2 hours
- SEC-F: Security Governance – 1 hour
- AL-E: Algorithms and Society – 1 hour
- DM-I: Data Security and Privacy – 1 hour

- FPL-H: Language Translation and Execution – 1 hour
- FPL-N: Runtime Behavior and Systems – 1 hour
- FPL-G: Type Systems – 1 hour
- HCI/Human Factors and Security – 1 hour
- NC-F: Network Security – 3 hours
- OS-G: Protection and Safety – 3 hours
- PDC-B: Communication – 1 hour
- PDC-D: Software Engineering – 2 hours
- SDF-A: Fundamental Programming Concepts and Practices – 1 hour
- SDF-D: Software Development Practices – 1 hour
- SE-F: Software Verification and Validation – 2 hours
- SEP-E: Privacy and Civil Liberties – 1 hour
- SEP-J: ecurity Policies, Laws and Computer Crime – 2 hours
- SF-G: Systems Security – 2 hours
- SPD-A: Common Aspects/Shared Concerns – 2 hours
- SPD-C: Mobile Platforms – 2 hours
- SPD-B: Web Platforms – 2 hours

The coverage exceeds 45 lecture hours, and so in a typical 3-credit semester course, instructors would need to decide what topics to emphasize and what not to cover without losing the perspective that the course should help students develop the *security mindset*.

Prerequisites:

- Depends on the selected topics

Skill statement:

- A student who completes this course should develop the security mindset and be ready to apply this mindset to problems to securing software and systems

Committee

Chair: Rajendra K. Raj, Rochester Institute of Technology, Rochester, NY, USA

Members:

- Vijay Anand, University of Missouri – St. Louis, MO, USA
- Diana Burley, American University, Washington, DC, USA
- Sherif Hazem, Central Bank of Egypt, Egypt
- Michele Maasberg, United States Naval Academy, Annapolis, MD, USA
- Sumita Mishra, Rochester Institute of Technology, Rochester, NY, USA
- Nicolas Sklavos, University of Patras, Patras, Greece

- Blair Taylor, Towson University, MD, USA
- Jim Whitmore, Dickinson College, Carlisle, PA, USA

Contributors:

- Markus Geissler, Cosumnes River College, CA, USA
- Daniel Zappala, Brigham Young University, UT, USA

Society, Ethics and Professionalism (SEP)

Preamble

While technical issues dominate the computing curriculum, they do not constitute a complete educational program in the broader context. Students must also be exposed to the larger societal context of computing to develop an understanding of the relevant social, ethical, legal 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 from CS1991 [1]:

> *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 (more specifically, how these advances are used by humans) continue to significantly impact the way we live and work, the critical importance of social and ethical issues and professional practice continues to increase in importance and consequence. The ways humans use computer-based products and platforms, while hopefully providing opportunities, also introduce ever more challenging problems each year. A recent example is the emergence of Generative AI including large language models that generate code. A 2020 *Communications of the ACM* article [2] stated: “because computing as a discipline is becoming progressively more entangled within the human and social lifeworld, computing as an academic discipline must move away from engineering-inspired curricular models and integrate the analytic lenses supplied by social science theories and methodologies.”

In parallel to a heightened awareness of the social consequences computing has on the world, computing communities have become much more aware - and active - in areas of Inclusion, Diversity, Equity and Accessibility. All computing students deserve an inclusive, diverse, equitable and accessible inclusive learning environment. However, computing students have a unique duty to ensure that when put to practice, their skills, knowledge, and competencies are applied in similar fashion, ethically and professionally, in the society they are in. For these reasons, inclusion, diversity, equity and accessibility are inherently a part of Society, Ethics, and Professionalism, and a new knowledge unit has been added that addresses this.

Computer science educators may opt to deliver the material in this knowledge area integrated into the context of traditional technical and theoretical courses, in dedicated courses (ideally a combination of both) and as special units as part of capstone, project, and professional practice courses. The material in this knowledge area is best covered through a combination of all the above. It is too commonly held that many topics in knowledge units listed as CS Core may not readily lend themselves to being covered in other more traditional computer science courses. However many of these topics naturally arise and others can be included with minimal effort, and the benefits of exposing students to these topics within the context of those traditional courses is most valuable. Nonetheless institutional challenges will present barriers; for instance some of these traditional courses may not be offered at a given institution and in such cases it is difficult to cover these topics appropriately without a dedicated course. However, if social, ethical and professional considerations are covered only in a dedicated course and not in the context of others, it could reinforce the false notion that technical processes are void of these important aspects, or that they are more isolated than they are in reality. Because of the broad relevance of these knowledge units, it is important that as many traditional courses as possible include aspects such as case studies that analyze ethical, legal, social and professional considerations in the context of the technical subject matter of those courses. Courses in areas such as software engineering, databases, computer graphics, computer networks, information assurance and security, and introduction to computing provide obvious context for analysis of such issues. However, an ethics-related module could be developed for almost any course in the curriculum. It would be explicitly against the spirit of these recommendations to have only a dedicated course. Further, these topics should be covered in courses starting from year 1. Presenting them as advanced topics in later courses only creates an artificial perception that SEP topics are only important at a certain level or complexity. While it is true that the importance and consequence of SEP topics *increases* with level and complexity, introductory topics are not devoid of SEP topics. Further, many SEP topics are *best* presented early to lay a foundation for more intricate topics later in the curriculum.

Running through all the issues in this area is the need to speak to the computing practitioner's responsibility to proactively address these issues by both ethical and technical actions. Today it is important not only for the topics in this knowledge area, but for students' knowledge in general, that the ethical issues discussed in any course should be directly related to - and arise naturally from - the subject matter of that course. Examples include a discussion in a database course of the societal, ethical and professional aspects of data aggregation or data mining, or a discussion in a software engineering course of the potential conflicts between obligations to the customer and users as well as all others affected by their work. 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, SIGCAS (ACM Special Interest Group on Computers and Society), and other organizations. Additionally, it is the educator's responsibility to impress upon students that this area is just as important - in ways more important - than technical areas. The societal, ethical, and professional knowledge gained in studying topics in this knowledge area will be used throughout one's career and are transferable between projects, jobs, and even industries, particularly as one's career progresses into project leadership and management.

The ACM Code of Ethics and Professional Conduct [3], the IEEE Code of Ethics [4], and the AAAI Code of Ethics and Professional Conduct [5] provide guidance that serve as the basis for the conduct of all computing professionals in their work. The ACM Code emphasizes that ethical reasoning is not an algorithm to be followed and computer professionals are expected to consider how their work impacts the public good as the primary consideration. It falls to computing educators to highlight the domain-specific role of these topics for our students, but programs should certainly be willing to lean heavily on complementary courses from the humanities and social sciences.

We observe that computing educators are not moral philosophers. Yet CS2023, as with past CS curricular recommendations, indicate the need for ethical analysis. CS2023 along with all previous CS Curricular reports are quite clear on the required mathematical foundations that students are expected to gain and the courses from mathematics departments that provide such training. Yet, the same is not true of moral philosophy. No one would expect a student to be able to provide a proof by induction until after having successfully completed a course in discrete mathematics. Yet the parallel with respect to ethical analyses is somehow absent. We seemingly do expect our students to perform ethical analysis without having the appropriate prerequisite knowledge from philosophy.

The lack of such prerequisite training has facilitated graduates operating with a certain ethical egoism (e.g., 'Here's what I believe/think/feel is right'). However, regardless of how well intentioned, one might conclude that this is what brought us to this point in history where computer crimes, hacks, scandals, data breaches, and the general misuse of computing technology (including the data it consumes and produces) is a frequent occurrence. Certainly, computing graduates who have learned how to apply the various ethical frameworks or lenses proposed through the ages would only serve to improve this situation. In retrospect, to ignore the lessons from moral philosophy, which have been debated and refined for millenia, on what it means to act justly, or work for the common good, appears as hubris.

Changes since CS 2013

- Inclusion of SEP/Inclusion, Diversity, Equity, and Accessibility knowledge unit
- Changed titles of two knowledge units (e.g. Professional Communication -> Communication, Analytical Tools -> Methods for Ethical Analysis)

[1] ACM/IEEE-CS Joint Curriculum Task Force, Computing Curricula 1991 (1991), ACM Press and IEEE Computer Society Press.

[2] Randy Connolly. 2020. Why computing belongs within the social sciences. Commun. ACM 63, 8 (August 2020), 54–59. https://doi.org/10.1145/3383444

[3] ACM Code of Ethics and Professional Conduct. www.acm.org/about/code-of-ethics

[4] IEEE Code of Ethics on Professional Activities.
https://www.ieee.org/about/corporate/governance/p7-8.html

[5] AAAI Code of Professional Ethics and Conduct. https://aaai.org/Conferences/code-of-ethics-and-conduct.php

Core Hours

Knowledge Units	CS Core	KA Core
Social Context	3	2
Methods for Ethical Analysis	2	1
Professional Ethics	2	2
Intellectual Property	1	1
Privacy and Civil Liberties	2	1
Communication	2	1
Sustainability	1	1
History	1	1
Economies of Computing	0	1
Security Policies, Laws and Computer Crimes	2	1
Equity, Diversity and Inclusion	2	2
Total	18	14

Knowledge Units

SEP-Context: Social Context

Computers, the internet, and artificial intelligence, perhaps more than any other technologies, have transformed society over the past several decades, with dramatic increases in human productivity; an explosion of options for news, entertainment, and communication; and fundamental breakthroughs in almost every branch of science and engineering. It is also imperative to recognize that this is not a one-way street. Society also affects computing, resulting in a complex socio-technical context that is constantly changing, requiring the perspective of history to put the present, as well as possible futures, into appropriate perspective.

Social Context provides the foundation for all other knowledge units in SEP, particularly Professional Ethics.

CS Core:

1. Social implications (e.g. political and cultural ideologies) in a hyper-networked world where the capabilities and impact of social media, artificial intelligence and computing in general are rapidly evolving
2. Impact of computing applications (e.g. social media, artificial intelligence applications) on individual well-being, and safety of all kinds (e.g., physical, emotional, economic)
3. Consequences of involving computing technologies, particularly artificial intelligence, biometric technologies and algorithmic decision-making systems, in civic life (e.g., facial recognition technology, biometric tags, resource distribution algorithms, policing software)
4. How deficits in diversity and accessibility in computing affect society and what steps can be taken to improve diversity and accessibility in computing

KA Core:

5. Growth and control of the internet, computing, and artificial intelligence
6. Often referred to as the digital divide, differences in access to digital technology resources and its resulting ramifications for gender, class, ethnicity, geography, and/or underdeveloped countries
7. Accessibility issues, including legal requirements and dark patterns
8. Context-aware computing

Illustrative Learning Outcomes:

CS Core:

1. Describe different ways that computer technology (networks, mobile computing, cloud computing) mediates social interaction at the personal and social group level.
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.
3. Interpret the social context of a given design and its implementation.
4. Evaluate the efficacy of a given design and implementation using empirical data.
5. Articulate the implications of social media use for different identities, cultures, and communities.

KA Core:

6. Explain the internet's role in facilitating communication between citizens, governments, and each other.
7. Analyze the effects of reliance on computing in the implementation of democracy (e.g., delivery of social services, electronic voting).
8. Describe the impact of the under-representation of people from historically minoritized populations in the computing profession (e.g., industry culture, product diversity).
9. Explain the implications of context awareness in ubiquitous computing systems.
10. Explain how access to the internet and computing technologies affect different societies.
11. Discuss why/how internet access can be viewed as a human right.

SEP-Ethical-Analysis: Methods for Ethical Analysis

Ethical theories and principles are the foundations of ethical analysis because they are the viewpoints which can provide guidance along the pathway to a decision. Each theory emphasizes different assumptions and methods for determining the ethicality of a given action. It is important for students to recognize that decisions in different contexts may require different ethical theories to arrive at ethically acceptable outcomes, and what constitutes 'acceptable' depends on a variety of factors such as cultural context. Applying methods for ethical analysis requires both an understanding of the underlying principles and assumptions guiding a given tool and an awareness of the social context for that decision. Traditional ethical frameworks as provided by western philosophy can be useful, but they are not all-inclusive. Effort must be taken to include decolonial, indigenous and historically marginalized ethical perspectives whenever possible. No theory will be universally applicable to all contexts, nor is any single ethical framework the 'best'. Engagement across various ethical schools of thought is important for students to develop the critical thinking needed in judiciously applying methods for ethical analysis of a given situation.

CS Core:

1. Avoiding fallacies and misrepresentation in argumentation
2. Ethical theories and decision-making (philosophical and social frameworks)
3. Recognition of the role culture plays in our understanding, adoption, design, and use of computing technology
4. Why ethics is important in computing, and how ethics is similar to, and different from, laws and social norms

KA Core:

5. Professional checklists
6. Evaluation rubrics
7. Stakeholder analysis
8. Standpoint theory
9. Introduction to ethical frameworks (e.g., consequentialism such as utilitarianism, non-consequentialism such as duty, rights or justice, agent-centered such as virtue or feminism, contractarianism, ethics of care) and their use for analyzing an ethical dilemma

Illustrative Learning Outcomes:

CS Core:

1. Recognize and describe how a given cultural context impacts decision making.
2. Illustrate the use of example and analogy in ethical argument.
3. Analyze (and avoid) basic logical fallacies in an argument.
4. Analyze an argument to identify premises and conclusion.
5. Evaluate how and why ethics is so important in computing and how it relates to cultural norms, values, and law.
6. Justify a decision made on ethical grounds.

KA Core:

7. Evaluate all stakeholder positions in relation to their cultural context in a given situation.

8. Evaluate the potential for introducing or perpetuating ethical debt (deferred consideration of ethical impacts or implications) in technical decisions.
9. Discuss the advantages and disadvantages of traditional ethical frameworks
10. Analyze ethical dilemmas related to the creation and use of technology from multiple perspectives using ethical frameworks

SEP-Professional-Ethics: Professional Ethics

Computer ethics is a branch of practical philosophy that deals with how computing professionals should make decisions regarding professional and social conduct. There are three primary influences: 1) The individual's own personal ethical code, 2) Any informal or formal code of ethical behavior existing in the workplace, applicable licensures or certifications, and 3) Exposure to formal codes of ethics and ethical frameworks.

CS Core:

1. Community values and the laws by which we live
2. The nature of professionalism including care, attention and discipline, fiduciary responsibility, and mentoring
3. Keeping up-to-date as a computing professional in terms of familiarity, tools, skills, legal and professional frameworks as well as the ability to self-assess and progress in the computing field
4. Professional certification, codes of ethics, conduct, and practice, such as the ACM/IEEE-CS, SE, AITP, IFIP and international societies
5. Accountability, responsibility and liability (e.g., software correctness, reliability and safety, warranty, negligence, strict liability, ethical approaches to security vulnerability disclosures)
6. Introduction to theories describing the human creation and use of technology including instrumentalism, sociology of technological systems, disability justice, neutrality thesis, pragmatism, and decolonial theories
7. Strategies for recognizing and reporting designs, systems, software, and professional conduct (or their outcomes) that may violate law or professional codes of ethics.

KA Core:

8. The role of the computing professional and professional societies in public policy
9. Maintaining awareness of consequences
10. Ethical dissent and whistle-blowing
11. The relationship between regional culture and ethical dilemmas
12. Dealing with harassment and discrimination
13. Forms of professional credentialing
14. Ergonomics and healthy computing environments
15. Time to market and cost considerations versus quality professional standards

Illustrative Learning Outcomes:

CS Core:

1. Identify ethical issues that arise in software design, development practices, and software deployment

2. Demonstrate how to address ethical issues in specific situations.
3. Explain the ethical responsibility of ensuring software correctness, reliability and safety including from where this responsibility arises (e.g., ACM/IEEE/AAAI Codes of Ethics, laws and regulations, organizational policies).
4. Describe the mechanisms that typically exist for a professional to keep up-to-date in ethical matters.
5. Describe the strengths and weaknesses of relevant professional codes as expressions of professionalism and guides to decision-making.
6. Analyze a global computing issue, observing the role of professionals and government officials in managing this problem.

KA Core:

8. Describe ways in which professionals and professional organizations may contribute to public policy.
9. Describe the consequences of inappropriate professional behavior.
10. Be familiar with whistleblowing and have access to knowledge to guide one through an incident.
11. Provide examples of how regional culture interplays with ethical dilemmas.
12. Discuss forms of harassment and discrimination and avenues of assistance.
13. Examine various forms of professional credentialing.
14. Explain the relationship between ergonomics in computing environments and people's health.
15. Describe issues associated with industries' push to focus on time to market versus enforcing quality professional standards.

SEP-IP: Intellectual Property

Intellectual property refers to a range of intangible rights of ownership in any product of the human intellect, such as a software program. Laws, which vary by country, provide different methods for protecting these rights of ownership based on their type. Ideally, intellectual property laws balance the interests of creators and of users of the property. There are essentially four types of intellectual property rights relevant to software: patents, copyrights, trade secrets and trademarks. Moreover, property rights are also often protected by user licenses. Each affords a different type of legal protection.

CS Core:

1. Intellectual property rights
2. Intangible digital intellectual property (IDIP)
3. Legal foundations for intellectual property protection
4. Common software licenses (e.g., MIT, GPL and its variants, Apache, Mozilla, Creative Commons)
5. Plagiarism and authorship

KA Core:

6. Philosophical foundations of intellectual property
7. Forms of intellectual property (e.g., copyrights, patents, trade secrets, trademarks) and the rights they protect

8. Limitations on copyright protections, including fair use and the first sale doctrine
9. Intellectual property laws and treaties that impact the enforcement of copyrights
10. Software piracy and technical methods for enforcing intellectual property rights, such as digital rights management and closed source software as a trade secret
11. Moral and legal foundations of the open source movement
12. Systems that use others' data (e.g., large language models)

Illustrative Learning Outcomes:

CS Core:

1. Describe and critique legislation and precedent aimed at digital copyright infringements.
2. Identify contemporary examples of intangible digital intellectual property.
3. Select an appropriate software license for a given project.
4. Justify legal and ethical uses of copyrighted materials.
5. Interpret the intent and implementation of software licensing.
6. Determine whether a use of copyrighted material is likely to be fair use.
7. Evaluate the ethical issues inherent in various plagiarism detection mechanisms.
8. Identify multiple forms of plagiarism beyond verbatim copying of text or software (e.g., intentional paraphrasing, authorship misrepresentation, and improper attribution).

KA Core:

9. Discuss the philosophical bases of intellectual property in an appropriate context (e.g., country, etc.).
10. Weigh the conflicting issues involved in securing software patents.
11. Characterize and contrast the protections and obligations of copyright, patent, trade secret, and trademarks.
12. Explain the rationale for the legal protection of intellectual property in the appropriate context (e.g., country, etc.).
13. Evaluate the use of copyrighted work under the concepts of fair use and the first sale doctrine.
14. Identify the goals of the open source movement and its impact on fields beyond computing, such as the right-to-repair movement.
15. Characterize the global nature of software piracy.
16. Critique the use of technical measures of digital rights management (e.g., encryption, watermarking, copy restrictions, and region lockouts) from multiple stakeholder perspectives.
17. Discuss the nature of anti-circumvention laws in the context of copyright protection.

SEP-Privacy: Privacy and Civil Liberties

Electronic information sharing highlights the need to balance privacy protections with information access. The ease of digital access to many types of data makes privacy rights and civil liberties more complex, especially given cultural and legal differences in these areas. Complicating matters further, privacy also has interpersonal, organizational, business, and governmental components. In addition, the interconnected nature of online communities raises challenges for managing expectations and protections for freedom of expression in various cultures and nations. Technology companies that provide platforms for user-generated content are under increasing pressure to perform governance tasks, potentially facing liability for their decisions.

CS Core:

1. Privacy implications of widespread data collection including but not limited to transactional databases, data warehouses, surveillance systems, and cloud computing
2. Conceptions of anonymity, pseudonymity, and identity
3. Technology-based solutions for privacy protection (e.g., end-to-end encryption and differential privacy)
4. Civil liberties and cultural differences

KA Core:

5. Philosophical and legal conceptions of the nature of privacy
6. Legal foundations of privacy protection in relevant jurisdictions (e.g., GDPR in the EU)
7. Privacy legislation in areas of practice (e.g., HIPAA in the US)
8. Freedom of expression and its limitations
9. User-generated content, content moderation, and liability

Illustrative Learning Outcomes:

CS Core:

1. Evaluate solutions to privacy threats in transactional databases and data warehouses.
2. Describe the role of data collection in the implementation of pervasive surveillance systems (e.g., RFID, face recognition, toll collection, mobile computing).
3. Distinguish the concepts and goals of anonymity and pseudonymity
4. Describe the ramifications of technology-based privacy protections, including differential privacy and end-to-end encryption
5. Identify cultural differences regarding the nature and necessity of privacy and other civil liberties.

KA Core:

6. Discuss the philosophical basis for the legal protection of personal privacy in an appropriate context (e.g., country, etc.).
7. Critique the intent, potential value and implementation of various forms of privacy legislation.
8. Identify strategies to enable appropriate freedom of expression.

SEP-Communication

Computing is an inherently collaborative and social discipline making communication an essential aspect of computing. Much but not all of this communication occurs in a professional setting where communication styles, expectations, and norms differ from other contexts where similar technology, such as email or messaging, might be used. Both professional and informal communication conveys information to various audiences who may have very different goals and needs for that information. It is also important to note that computing professionals are not just communicators, but are also listeners who must be able to hear and thoughtfully make use of feedback received from various stakeholders. Effective communication skills are not something one ‘just knows’ - they are developed and can be learned. Communication skills are best taught in context throughout the undergraduate curriculum.

CS Core:

1. Interpreting, summarizing, and synthesizing technical material, including source code and documentation
2. Writing effective technical documentation and materials (tutorials, reference materials, API documentation)
3. Identifying, describing, and employing (clear, polite, concise) oral, written, and electronic team and group communication.
4. Understanding and enacting awareness of audience in communication by communicating effectively with different stakeholders such as customers, leadership, or the general public
5. Appropriate and effective team communication including utilizing collaboration tools and conflict resolution
6. Recognizing and avoiding the use of rhetorical fallacies when resolving technical disputes
7. Understanding accessibility and inclusivity requirements for addressing professional audiences

KA Core:

8. Demonstrating cultural competence in written and verbal communication
9. Using synthesis to concisely and accurately convey tradeoffs in competing values driving software projects including technology, structure/process, quality, people, market and financial
10. Use writing to solve problems or make recommendations in the workplace, such as raising ethical concerns or addressing accessibility issues

Illustrative Learning Outcomes:

CS Core:

1. Understand the importance of writing concise and accurate technical documents following well-defined standards for format and for including appropriate tables, figures, and references.
2. Evaluate written technical documentation for technical accuracy, concision, lack of ambiguity, and awareness of audience.
3. Develop and deliver an audience aware, accessible, and organized formal presentation.
4. Plan interactions (e.g., virtual, face-to-face, shared documents) with others in ways that invite inclusive participation, model respectful consideration of others’ contributions, and explicitly value diversity of ideas.
5. Recognize and describe qualities of effective communication (e.g., virtual, face-to-face, intragroup, shared documents).
6. Understand how to effectively and appropriately communicate as a member of a team including conflict resolution techniques.

KA Core:

7. Discuss ways to influence performance and results in diverse and cross-cultural teams.

8. Evaluate personal strengths and weaknesses to work remotely as part of a team drawing from diverse backgrounds and experiences.

SEP-Sustainability

Sustainability is defined by the United Nations as “development that meets the needs of the present without compromising the ability of future generations to meet their own needs."[1] Alternatively, it is the “balance between the environment, equity and economy.”[2] As computing extends into more and more aspects of human existence, we are already seeing estimates that 10% of global electricity usage is spent on computing, and that percentage will continue growing. Further, electronics contribute individually to demand for rare earth elements, mineral extraction, and countless e-waste concerns. Students should be prepared to engage with computing with a background that recognizes these global and environmental costs and their potential long term effects on the environment and local communities.

[1] https://www.un.org/en/academic-impact/sustainability

[2] https://www.sustain.ucla.edu/what-is-sustainability

CS Core:

1. Being a sustainable practitioner by taking into consideration environmental, social, and cultural impacts of implementation decisions (e.g., sustainability goals, algorithmic bias/outcomes, economic viability, and resource consumption)
2. Local/regional/global social and environmental impacts of computing systems use and disposal (e.g. carbon footprints, e-waste) in hardware (e.g., data centers) and software (e.g. blockchain, AI model training and use).
3. Discuss the tradeoffs involved in proof-of-work and proof-of-stake algorithms

KA Core:

4. Guidelines for sustainable design standards
5. Systemic effects of complex computer-mediated phenomena (e.g., social media, offshoring, remote work)
6. 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
7. Conduct research on applications of computing to environmental issues, such as energy, pollution, resource usage, recycling and reuse, food management / production, and others
8. How the sustainability of software systems are interdependent with social systems, including the knowledge and skills of its users, organizational processes and policies, and its societal context (e.g., market forces, government policies)

Illustrative Learning Outcomes:

CS Core:

1. Identify ways to be a sustainable practitioner.
2. For any given project (software artifact, hardware, etc.) evaluate the environmental impacts of its deployment. (e.g., energy consumption, contribution to e-waste, impact of manufacturing).
3. Illustrate global social and environmental impacts of computer use and disposal (e-waste).
4. List the sustainable effects of modern practices and activities (e.g., remote work, online commerce, cryptocurrencies, data centers).

KA Core:

5. 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.
6. Investigate the social and environmental impacts of new system designs.
7. Identify guidelines for sustainable IT design or deployment.
8. Investigate pervasive computing in areas such as smart energy systems, social networking, transportation, agriculture, supply-chain systems, environmental monitoring and citizen activism.
9. Assess computing applications in respect to environmental issues (e.g., energy, pollution, resource usage, recycling and reuse, food management and production).

SEP-History

History is important because it provides a mechanism for understanding why our computing systems operate the way they do, the societal contexts in which these approaches arose, and how those continue to echo through the discipline today. This history of computing is taught to provide a sense of how the rapid change in computing impacts society on a global scale. It is often taught in context with foundational concepts, such as system fundamentals and software development fundamentals.

CS Core:

1. The history of computing: hardware, software, and human/organizational and the role of this in present social contexts

KA Core:

2. Age I: Prehistory—the world before ENIAC (1946): Ancient analog computing (Stonehenge, Antikythera mechanism, Salisbury Cathedral clock, etc.), human-calculated number tables, Euclid, Lovelace, Babbage, Gödel, Church, Turing, pre-electronic (electro-mechanical and mechanical) hardware
3. Age II: Early modern (digital) computing - ENIAC, UNIVAC, Bombes (Bletchley Park codebreakers), computer companies (e.g., IBM), mainframes, etc.
4. Age III: Modern (digital) computing - PCs, modern computer hardware and software, Moore's Law
5. Age IV: Internet - networking, internet architecture, browsers and their evolution, standards, big players (Google, Amazon, Microsoft, etc.), distributed computing
6. Age V: Cloud - smartphones (Apple, Android, and minor ones), cloud computing, remote servers, software as a service (SaaS), security and privacy, social media
7. Age VI: Emerging AI-assisted technologies including decision making systems, recommendation systems, generative AI and other machine learning driven tools and technologies

Illustrative Learning Outcomes:

CS Core:

1. Understand the relevance and impact of computing history on recent events, present context, and possible future outcomes. *Ideally from more than one cultural perspective.*

KA Core:

2. Identify significant trends in the history of the computing field.
3. Identify the contributions of several pioneering individuals or organizations (research labs, computer companies, government offices) in the computing field.
4. Discuss the historical context for important moments in history of computing, such as the move from vacuum tubes to transistors (TRADIC), early seminal operating systems (e.g., OS 360), Xerox PARC and the first Apple computer with a GUI, the creation of specific programming language paradigms, the first computer virus, the creation of the internet, the creation of the WWW, the dot com bust, Y2K, the introduction of smartphones, etc.
5. Compare daily life before and after the advent of personal computers and the Internet.

SEP-Economies: Economies of Computing

The economies of computing are important to those who develop and provide computing resources and services to others as well as society in general. They are equally important to users of these resources and services, both professional and non-professional.

KA Core:

1. Economies of providers: regulated and unregulated, monopolies, network effects, and open-market. “Walled Gardens” in tech environments
2. The knowledge and attention economies
3. Effect of skilled labor supply and demand on the quality of computing products
4. Pricing strategies in the computing domain: subscriptions, planned obsolescence, software licenses, open-source, free software
5. Outsourcing and off-shoring software development; impacts on employment and on economics
6. Consequences of globalization for the computer science profession and users
7. Differences in access to computing resources and the possible effects thereof
8. Automation and its effect on job markets, developers, and users
9. Economies of scale, startups, entrepreneurship, philanthropy
10. How computing is changing personal finance: Blockchain and cryptocurrencies, mobile banking and payments, SMS payment in developing regions, etc.

Illustrative Learning Outcomes:

KA Core:

1. Summarize concerns about monopolies in tech, walled gardens vs open environments, etc.
2. Identify several ways in which the information technology industry and users are affected by shortages in the labor supply.
3. Outline the evolution of pricing strategies for computing goods and services.
4. Explain the social effects of the knowledge and attention economies.

5. Summarize the consequences of globalization and nationalism in the computing industry.
6. Describe the effects of automation on society, and job markets in particular.
7. Detail how computing has changed the corporate landscape
8. Outline how computing has changed personal finance and the consequences of this, both positive and negative.

SEP-Security: Security Policies, Laws and Computer Crimes

While security policies, laws and computer crimes are important, it is essential they are viewed with the foundation of other Social and Professional knowledge units, such as Intellectual Property, Privacy and Civil Liberties, Social Context, and Professional Ethics. Computers, the internet, and artificial intelligence, perhaps more than any other technologies, have transformed society over the past 75 years. At the same time, they have contributed to unprecedented threats to privacy; new categories of computer crime and anti-social behavior; major disruptions to organizations; and the large-scale concentration of risk into information systems.

CS Core:

1. Examples of computer crimes and legal redress for computer criminals
2. Social engineering, computing-enabled fraud, and recovery
3. Identify what constitutes computer crime, such as Issues surrounding the misuse of access and breaches in security
4. Motivations and ramifications of cyber terrorism and criminal hacking, “cracking”
5. Effects of malware, such as viruses, worms and Trojans
6. Attacks on critical infrastructure such as electrical grids and pipelines

KA Core:

7. Benefits and challenges of existing and proposed computer crime laws
8. Security policies and the challenges of compliance
9. Responsibility for security throughout the computing life cycle
10. International and local laws and how they intersect

Illustrative Learning Outcomes:

CS Core:

1. List classic examples of computer crimes and social engineering incidents with societal impact.
2. Identify laws that apply to computer crimes.
3. Describe the motivation and ramifications of cyber terrorism, data theft, hacktivism (hacking as activism), ransomware, and other attacks..
4. Examine the ethical and legal issues surrounding the misuse of access and various breaches in security.
5. Discuss the professional's role in security and the trade-offs involved.

KA Core:

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.
7. Draft a company-wide security policy, which includes procedures for managing passwords and employee monitoring.
8. Understand how legislation from one region may affect activities in another (e.g. how EU GDPR applies globally, when EU persons are involved).

SEP-IDEA: Inclusion, Diversity, Equity, and Accessibility

Computer Science has had—since its inception as a field—a diversity problem. Despite being a creative, highly compensated field with myriad job (and other) opportunities, racial, gender and other inequities in representation are pervasive. For too many students, their first computer science course is their last. There are many factors including the legacy of systemic racism, ableism, sexism, classism, and other injustices that contribute to the lack of diverse identities within computer science, and there is no single, quick fix.

CS2023's sponsoring organizations are ACM, IEEE CS, and AAAI. Each of those organizations [https://www.acm.org/diversity-inclusion/about#DEIPrinciples, https://www.ieee.org/about/diversity-index.html, https://aaai.org/Organization/diversity-statement.php] place a high value on inclusion, diversity, equity, and accessibility; and our computer science classrooms should promote and model those principles. We should welcome and seek diversity—the gamut of human differences including gender, gender identity, race, politics, ability and attributes, religion, nationality, etc.—in our classrooms, departments and campuses. We should strive to make our classrooms, labs, and curricula accessible and to promote inclusion; the sense of belonging we feel in a community where we are respected and wanted. To achieve equity, we must allocate resources, promote fairness, and check our biases to ensure persons of all identities achieve success. Accessibility should be considered and implemented in all computing activities and products.

Explicitly infusing inclusion, diversity, equity, and accessibility across the computer science curriculum demonstrates its importance for the department, institution, and our field—all of which likely have a IDEA statement and/or initiative(s). This emphasis on IDEA is important ethically and a bellwether issue of our time. Many professionals in computing already recognize attention to diversity, equity, inclusion, accessibility as integral parts of disciplinary practice. Regardless of the degree to which IDEA values appear in any one computer science class, research suggests that a lack of attention to IDEA will result in inferior designs. Not only does data support that diverse teams outperform homogeneous ones, but diverse teams may have prevented egregious technology failures in the headlines such as facial recognition misuse, airbag injuries, and deaths.

CS Core:

1. How identity impacts and is impacted by computing environments (academic and professional) and technologies
2. The benefits of diverse development teams and the impacts of teams that are not diverse.
3. Inclusive language and charged terminology, and why their use matters
4. Inclusive behaviors and why they matter
5. Designing and developing technology with accessibility in mind

6. Designing for accessibility
7. How computing professionals can influence and impact inclusion, diversity, equity, and accessibility both positively and negatively, not only through the software they create.

KA Core:

8. Highlight experts (practitioners, graduates, and upper level students) who reflect the identities of the classroom and the world
9. Benefits of diversity and harms caused by a lack of diversity
10. Historic marginalization due to technological supremacy and global infrastructure challenges to equity and accessibility

Illustrative Learning Outcomes:

CS Core:

1. Define and distinguish equity, equality, diversity, inclusion, and accessibility..
2. Describe the impact of power and privilege in the computing profession as it relates to culture, industry, products, and society.
3. Identify language, practices, and behaviors that may make someone feel included in a workplace and/or a team, and why is it relevant. Avoid charged terminology - see *Words Matter* (https://www.acm.org/diversity-inclusion/words-matter).
4. Evaluate the accessibility of your classroom or lab. Evaluate the accessibility of your webpage. (See https://www.w3.org/WAI/.)
5. Work collegially and respectfully with team members who do not share your identity. *It is not enough to merely assign team projects. Faculty should prepare students for teamwork and monitor, mentor, and assess the effectiveness of their student teams throughout a project.*
6. Compare the demographics of your institution’s computer science and STEM majors to the overall institutional demographics. If they differ, identify factors that contribute to inequitable access, engagement, and achievement in computer science among marginalized groups.
7. Compare the demographics of your institution to the overall community demographics. If they differ, identify factors that contribute to inequitable access, engagement, and achievement among marginalized groups.
8. Identify developers’ assumptions and values embedded in hardware and software design, especially as they pertain to usability by diverse populations.

KA Core:

9. Highlight experts (practitioners, graduates, and upper level students—current and historic) who reflect the identities of the classroom and the world.
10. Identify examples of the benefits that diverse teams can bring to software products, and those where a lack of diversity have costs.
11. Give examples of systemic changes that could positively address diversity, equity, and inclusion in a familiar context (i.e. in an introductory computing course).

Professional Dispositions

- Critical Self-reflection - Being able to inspect one's own actions, thoughts, biases, privileges, and motives will help in discovering places where professional activity is not up to current standards. Understand both conscious and unconscious bias and continuously work to counteract them.
- Responsiveness - Ability to quickly and accurately respond to changes in the field and adapt in a professional manner, such as shifting from in-person office work to remote work at home. These shifts require us to rethink our entire approach to what is considered "professional".
- Proactiveness - Being professional in the workplace means finding new trends (e.g. in accessibility or inclusion) and understanding how to implement them immediately for a more professional working environment.
- Cultural Competence - Prioritize cultural competence—the ability to work with people from cultures different from your own—by using inclusive language, watching for and counteracting conscious and unconscious bias, and encouraging honest and open communication.
- Advocation - Thinking, speaking and acting in ways that foster and promote inclusion, diversity, equity and accessibility in all ways including but not limited to teamwork, communication, and developing products (hardware and software).

Course Packaging Suggestions

In computing, Societal, Ethical, and Professional topics arise in all other knowledge areas and therefore should arise in the context of other computing courses, not just siloed in an "SEP course". These topics should be covered in courses starting from year 1(the only likely exception is SEP-Ethical-Analysis: Methods for Ethical Analysis) which could be delivered as part of a first-year course or via a seminar or an online independent study.

Presenting SEP topics as advanced topics only covered in later courses could create the incorrect perception that SEP topics are only important at a certain level or complexity. While it is true that the importance and consequence of SEP topics *increases* with level and complexity, introductory topics are not devoid of SEP topics. Further, many SEP topics are *best* presented early to lay a foundation for more intricate topics later in the curriculum.

Who should teach some of these topics is a complex topic. When SEP topics arise in other courses these are naturally often taught by the instructor teaching that course, although at times bringing in expert educators from other disciplines (e.g., law, ethics, etc.) could be advantageous. Stand-alone courses in SEP could be taught by a team of CS and other disciplines - although more logistically complicated, this may be a better approach than being taught by a single CS instructor. Regardless, who teaches SEP topics and/or courses warrants

At a minimum the SEP CS Core learning outcomes* are best covered in the context of courses covering other knowledge areas - ideally the SEP KA Core hours are also. *With the likely exception of SEP-Ethical Analysis: Methods for Ethical Analysis which could be delivered as discussed above.

At some institutions an **in-depth dedicated course** at the mid- or advanced-level may be offered covering all recommended topics in both the CS Core and KA Core knowledge units in close coordination with learning outcomes best covered in the context of courses covering other knowledge areas. Such a course could include:

- SEP-Context (5 hours)
- SEP-Ethical-Analysis: Methods for Ethical Analysis (3 hours)
- SEP-Professional-Ethics: Professional Ethics (4 hours)
- SEP-IP (2 hours)
- SEP-Privacy: Privacy and Civil Liberties (3 hours)
- SEP-Communication (3 hours)
- SEP-Sustainability (2 hours)
- SEP-History (2 hours)
- SEP-Economies: Economies of Computing (1 hour)
- SEP-Security: Security Policies, Laws and Computer Crimes (3 hours)
- SEP-IDEA: Diversity, Equity, and Inclusion (4 hours)

At some institutions a **dedicated minimal course** may be offered covering the CS Core knowledge units in close coordination with learning outcomes best covered in the context of courses covering other knowledge areas. Such a course could include:

- SEP-Context (3 hours)
- SEP-Ethical-Analysis: Methods for Ethical Analysis (2 hours)
- SEP-Professional-Ethics (2 hours)
- SEP-IP (1 hour)
- SEP-Privacy: Privacy and Civil Liberties (2 hours)
- SEP-Communication (2 hours)
- SEP-Sustainability (1 hour)
- SEP-History (1 hour)
- SEP-Security: Security Policies, Laws and Computer Crimes (2 hours)
- SEP-IDEA: Inclusion, Diversity, Equity, and Accessibility (2 hours)

References

1. Emanuelle Burton, Judy Goldsmith, Nicholas Mattei, Cory Siler, and Sara-Jo Swiatek. 2023. Teaching Computer Science Ethics Using Science Fiction. In Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 2 (SIGCSE 2023). Association for Computing Machinery, New York, NY, USA, 1184. https://doi.org/10.1145/3545947.3569618
2. Randy Connolly. 2020. Why computing belongs within the social sciences. Commun. ACM 63, 8 (August 2020), 54–59. https://doi.org/10.1145/3383444
3. Casey Fiesler. Tech Ethics Curricula: A Collection of Syllabi Used to Teach Ethics in Technology Across Many Universities
    a. https://cfiesler.medium.com/tech-ethics-curricula-a-collection-of-syllabi-3eedfb76be18
    b. Tech Ethics Curricula
4. Casey Fiesler. Tech Ethics Readings: A Spreadsheet of Readings Used to Teach Ethics in Technology Tech Ethics Class Readings
5. Stanford Embedded EthiCS, Embedding Ethics in Computer Science. https://embeddedethics.stanford.edu/
6. Jeremy, Weinstein, Rob Reich, and Mehran Sahami. *System Error: Where Big Tech Went Wrong and How We Can Reboot*. Hodder Paperbacks, 2023.
7. Baecker, R. Computers in Society: Modern Perspectives, Oxford University Press. (2019).

8. Embedded EthiCS @ Harvard: bringing ethical reasoning into the computer science curriculum. https://embeddedethics.seas.harvard.edu/about

Systems Fundamentals (SF)

Preamble

A computer system is a set of hardware and software infrastructures upon which applications are constructed. Computer systems have become a pillar of people's daily life. As such, learning the knowledge about computer systems, grasping the skills to use and design these systems, and understanding the fundamental rationale and principles in computer systems are essential to equip students with the necessary competency toward a career related to computer science.

In the curriculum of computer science, the study of computer systems typically spans across multiple courses, including, but not limited to, operating systems, parallel and distributed systems, communications networks, computer architecture and organization and software engineering. The System Fundamentals knowledge area, as suggested by its name, focuses on the fundamental concepts in computer systems that are shared by these courses within their respective cores. The goal of this knowledge area is to present an integrative 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 domain area. These concepts include an overview of computer systems, basic concepts such as state and state transition, resource allocation and scheduling, and so on.

Changes since CS 2013

Compared to CS2013, the SF knowledge area makes the following major changes to the knowledge units:

1. Added two new units: system security and system design;
2. Added a new unit of system performance, which includes the topics from the deprecated unit of proximity and the deprecated unit of virtualization and isolation;
3. Added a new unit of performance evaluation, which includes the topics from the deprecated unit of evaluation and the deprecated unit of quantitative evaluation;
4. Changed the unit of computational paradigms to overview of computer systems, deprecated some topics in the unit, and added topics from the deprecated unit of cross-layer communications;
5. Changed the unit of state and state transition to basic concepts, and added topics such as finite state machines;
6. Changed some topics in the unit of parallelism, such as simple application-level parallel processing;
7. Deprecated the unit of cross-layer communications, and moved parts of its topics to the unit of overview of computer systems;
8. Deprecated the units of evaluation and quantitative evaluation, and moved parts of their topics to the unit of performance evaluation;
9. Deprecated the units of proximity and virtualization and isolation, and moved parts of their topics to the unit of system performance;
10. Deprecated the units of parallelism, and moved parts of its topic to the unit of basic concepts;

11. Renamed the unit of reliability through redundancy to system reliability.

Core Hours

Knowledge Unit	**CS Core**	**KA Core**
Overview of Computer Systems	3	
Basic Concepts	4	
Resource Allocation and Scheduling	1	2
System Performance	2	2
Performance Evaluation	2	2
System Reliability	2	1
System Security	2	1
System Design	2	1
Total	**18**	**9**

Knowledge Units

SF-A: Overview of Computer Systems

***CS Core**:*

1. Basic building blocks and components of a computer (gates, flip-flops, registers, interconnections; datapath + control + memory)
2. Hardware as a computational paradigm: Fundamental logic building blocks; Logic expressions, minimization, sum of product forms
3. Programming abstractions, interfaces, use of libraries
4. Distinction between application and OS services, remote procedure call
5. Application-OS interaction
6. Basic concept of pipelining, overlapped processing stages
7. Basic concept of scaling: going faster vs. handling larger problems

Learning Outcomes:

1. Describe the basic building blocks of computers and their role in the historical development of computer architecture.
2. Design a simple logic circuit using the fundamental building blocks of logic design to solve a simple problem (e.g., adder).
3. Use tools for capture, synthesis, and simulation to evaluate a logic circuit design.
4. 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.
5. Describe that hardware, OS, VM, application are additional layers of interpretation/processing.

6. Describe the mechanisms of how errors are detected, signaled back, and handled through the layers.
7. Construct a simple program (e.g., a TCP client/server) using methods of layering, error detection and recovery, and reflection of error status across layers.
8. Find bugs in a layered program by using tools for program tracing, single stepping, and debugging.
9. Understand 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 simple, real-world examples.

SF-B: Basic Concepts

CS Core:

1. Digital vs. Analog/Discrete vs. Continuous Systems
2. Simple logic gates, logical expressions, Boolean logic simplification
3. Clocks, State, Sequencing
4. State and state transition (e.g., starting state, final state, life cycle of states)
5. Finite state machines (e.g., NFA, DFA)
6. Combinational Logic, Sequential Logic, Registers, Memories
7. Computers and Network Protocols as examples of State Machines
8. Sequential vs. parallel processing
9. Application-level sequential processing: single thread
10. Simple application-level parallel processing: request level (web services/client-server/distributed), single thread per server, multiple threads with multiple servers, pipelining

Learning Outcomes:

1. Describe the differences between digital and analog systems, and between discrete and continuous systems. Can give real-world examples of these systems.
2. Describe computations as a system characterized by a known set of configurations with transitions from one unique configuration (state) to another (state).
3. Describe the distinction between systems whose output is only a function of their input (stateless) and those with memory/history (stateful).
4. Develop state machine descriptions for simple problem statement solutions (e.g., traffic light sequencing, pattern recognizers).
5. Describe a computer as a state machine that interprets machine instructions.
6. 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.
7. Derive time-series behavior of a state machine from its state machine representation (e.g., TCP connection management state machine).
8. Write a simple sequential problem and a simple parallel version of the same program.
9. Evaluate the performance of simple sequential and parallel versions of a program with different problem sizes, and be able to describe the speed-ups achieved.
10. Demonstrate on an execution timeline that parallelism 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.

SF-C: Resource Allocation and Scheduling

CS Core:

1. Different types of resources (e.g., processor share, memory, disk, net bandwidth)
2. Common scheduling algorithms (e.g., first-come-first-serve scheduling, priority-based scheduling, fair scheduling and preemptive scheduling)

KA Core:

1. Advantages and disadvantages of common scheduling algorithms

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.
2. Describe how common scheduling algorithms work.
3. Describe the pros and cons of common scheduling algorithms
4. Implement common scheduling algorithms, and evaluate their performances.

SF-D: System Performance

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

CS Core:

1. Latencies in computer systems
    - Speed of light and computers (one foot per nanosecond vs. one GHz clocks)
    - Memory vs. disk latencies vs. across the network memory
2. Caches and the effects of spatial and temporal locality on performance in processors and systems
3. Caches and cache coherency in databases, operating systems, distributed systems, and computer architecture
4. Introduction into the processor memory hierarchy and the formula for average memory access time

KA Core:

1. Rationale of virtualization and isolation: protection and predictable performance
2. Levels of indirection, illustrated by virtual memory for managing physical memory resources
3. Methods for implementing virtual memory and virtual machines

Learning Outcomes:

1. Explain the importance of locality in determining system performance.
2. Calculate average memory access time and describe the tradeoffs in memory hierarchy performance in terms of capacity, miss/hit rate, and access time.
3. Explain why it is important to isolate and protect the execution of individual programs and environments that share common underlying resources.
4. 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.
5. Measure the performance of two application instances running on separate virtual machines, and determine the effect of performance isolation.

SF-E: Performance Evaluation

CS Core:

1. Performance figures of merit
2. Workloads and representative benchmarks, and methods of collecting and analyzing performance figures of merit
3. CPI (Cycles per Instruction) equation as tool for understanding tradeoffs in the design of instruction sets, processor pipelines, and memory system organizations.
4. Amdahl's Law: the part of the computation that cannot be sped up limits the effect of the parts that can
5. Order of magnitude analysis (Big O notation)
6. Analysis of slow and fast paths of a system
7. Events on their effect on performance (e.g., instruction stalls, cache misses, page faults)

KA Core:
1. Analytical tools to guide quantitative evaluation
2. Understanding layered systems, workloads, and platforms, their implications for performance, and the challenges they represent for evaluation
3. Microbenchmarking pitfalls

***Learning Outcomes**:*
1. Explain how the components of system architecture contribute to improving its performance.
2. Explain the circumstances in which a given figure of system performance metric is useful.
3. Explain the usage and inadequacies of benchmarks as a measure of system performance.
4. Describe Amdahl's law and discuss its limitations.
5. Use limit studies or simple calculations to produce order-of-magnitude estimates for a given performance metric in a given context.
6. Use software tools to profile and measure program performance.
7. Design and conduct a performance-oriented experiment of a common system (e.g., an OS and Spark).
8. Conduct a performance experiment on a layered system to determine the effect of a system parameter on system performance.

SF-F: System Reliability

***CS Core**:*
1. Distinction between bugs and faults
2. Reliability through redundancy: check and retry
3. Reliability through redundancy: redundant encoding (error correction codes, CRC, FEC)
4. Reliability through redundancy: duplication/mirroring/replicas

KA Core:
Other approaches to reliability

***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).

2. Articulate the distinction between detecting, handling, and recovering from faults, and the methods for their implementation.
3. Describe the role of error correction codes in providing error checking and correction techniques in memories, storage, and networks.
4. Apply simple algorithms for exploiting redundant information for the purposes of data correction.
5. Compare different error detection and correction methods for their data overhead, implementation complexity, and relative execution time for encoding, detecting, and correcting errors.

SF-G: System Security

***CS Core**:*

1. Common system security issues (e.g., virus, denial-of-service attack and eavesdropping)
2. Countermeasures
    - Cryptography
    - Security architecture

KA Core:

1. Countermeasures
    - Intrusion detection systems, firewalls

***Learning Outcomes**:*

1. Describe some common system security issues and give examples.
2. Describe some countermeasures against system security issues.

SF-H: System Design

***CS Core**:*

1. Common criteria of system design (e.g., liveness, safety, robustness, scalability and security)

KA Core:

1. Designs of representative systems (e.g., Apache web server, Spark and Linux)

***Learning Outcomes**:*

1. Describe common criteria of system design.
2. Given the functionality requirements of a system and its key design criteria, provide a high-level design of this system.
3. Describe the design of some representative systems.

Professional Dispositions

- **Meticulousness:** students must pay attention to details of different perspectives when learning about and evaluating systems.

- **Adaptiveness**: students must be flexible and adaptive when designing systems. Different systems have different requirements, constraints and working scenarios. As such, they require different designs. Students must be able to make appropriate design decisions correspondingly.

Math Requirements

Required:

- Discrete Math:
  - Sets and relations
  - Basic graph theory
  - Basic logic
- Linear Algebra:
  - Basic matrix operations
- Probability and Statistics
  - Random variable
  - Bayes theorem
  - Expectation and Variation
  - Cumulative distribution function and probability density function

Desirable:

- Basic queueing theory
- Basic stochastic process

Shared Topics and Crosscutting Themes

Shared Topics:

- Networking and communication (NC) –
- Operating system (OS)
- Architecture and organization (AR)
- Parallel and distributed computing (PDC).

Course Packaging Suggestions

Introductory Course to include the following:

- SF-A: Overview of Computer Systems - 2 hours
- SF-B: Basic Concepts - 6 hours
- SF-C: Resource Allocation and Scheduling - 4 hours
- SF-D: System Performance - 6 hours
- SF-E: Performance Evaluation - 6 hours
- SF-F: System Reliability - 4 hours

- SF-G: System Security - 6 hours
- SF-H: System Design - 6 hours

Pre-requisites:

- Sets and relations, basic graph theory and basic logic from Discrete Math
- Basic matrix operations from Linear Algebra
- Random variable, Bayes theorem, expectation and variation, cumulative distribution function and probability density function from Probability and Statistics

Skill statement: A student who completes this course should be able to (1) understand the fundamental concepts in computer systems; (2) understand the key design principles, in terms of performance, reliability and security, when designing computer systems; (3) deploy and evaluate representative complex systems (e.g., MySQL and Spark) based on their documentations, and (4) design and implement simple computer systems (e.g., an interactive program, a simple web server, and a simple data storage system).

Advanced Course to include the following:

- SF-H: System Design - 10 hours
- KUs from OS - 2 hours
- KUs from NC - 2 hours
- KUs from PDC - 2 hours
- KUs from AR - 2 hours
- SF-F: System Reliability - 10 hours
- SF-D: System Performance - 6 hours
- SF-G: System Security - 6 hours

Pre-requisites:

- Basic queueing theory and stochastic process
- Introductory Course of the SF KA

Skill statement: A student who completes this course should be able to (1) have a deeper understanding in the key design principles of computer system design, (2) map such key principles to the designs of classic systems (e.g., Linux, SQL and TCP/IP network stack) as well as that of more recent systems (e.g., Hadoop, Spark and distributed storage systems), and (3) design and implement more complex computer systems (e.g., a file system and a high-performance web server).

Committee

Chair: Qiao Xiang, Xiamen University, Xiamen, China

Members:

- Doug Lea, State University of New York at Oswego, Oswego, NY, USA
- Monica D. Anderson, University of Alabama, Tuscaloosa, AL, USA
- Matthias Hauswirth, University of Lugano, Lugano, Switzerland
- Ennan Zhai, Alibaba Group, Hangzhou, China

- Yutong Liu, Shanghai JiaoTong University, Shanghai, China

Contributors:

- Michael S. Kirkpatrick, James Madison University, Harrisonburg, VA, USA
- Linghe Kong, Shanghai JiaoTong University, Shanghai, China

Specialized Platform Development (SPD)

Preamble

The Specialized Platform Development (SPD) Knowledge Area (KA) refers to attributes involving creating a software platform to address specific needs and requirements for particular areas. Specialized platforms, such as healthcare providers, financial institutions, or transportation companies, are tailored to meet specific needs. Developing a specialized platform typically involves several key stages, i.e., requirements, design, development, deployment, and maintenance.

The SPD Beta Versions considered the following factors:

- ***Emerging Computing Areas*** such as data science/analytics – use multi- platforms to retrieve sensing data. Cybersecurity – involves protecting certain data extraction, recognizing protocols to protect network transfer ability, and manipulating it. Artificial intelligence and machine learning – use artifacts that retrieve information for robotics and drones to perform specific tasks. This continuous emergence of computing technology areas has increased the appetite to develop platforms communicating software with specialized environments. This need has also increased the need to develop specialized programming languages for these platforms, such as web and mobile development. The Interactive Computing Platform addresses the advent of Large Language Models (LLMs), such as OpenAI's ChatGPT, OpenAI's Codex, and GitHub's Copilot, in addition to other platforms that perform data analysis, and visualizations.
- **Industry needs and competencies** have created a high demand for developers on specialized platforms, such as mobile, web, robotics, embedded, and interactive. Some of the unique professional competencies obtained by current job descriptions relevant to this KA are:
    - *Create a mobile app that provides a consistent user experience across various devices, screen sizes, and operating systems.*
    - *Analyze people's experience using a novel peripheral for an immersive system facilitated using a head-mounted display and mixed reality, with attention to usability and accessibility specifications.*
    - *Build and optimize a secure web page for evolving business needs using a variety of appropriate programming languages.*
    - *Develop application programming interfaces (APIs) to support mobile functionality and remain current with the terminology, concepts, and best practices for coding mobile apps.*
    - *Availability of devices and artifacts, such as raspberry PIs, Arduinos, and mobile devices. The low cost of microcontrollers and devices, such as robots using ROS that can perform specialized actions*

Changes since CS 2013

The consideration of these factors resulted in the following significant changes from the CS2013 version:

- **Renamed of the Knowledge Area name**: From Platform-Based Development (PBD) to Specialized Platform Development due to the specific needs of the already mentioned tasks. This

particular KA is often called *software platform development* since the specialized development takes part in the software stages for multi-platform development.
- **Increase of Computer Science Core Hours**: Based on the already mentioned needs, the SPD beta version has increased the number of computer science course hours from 0 to 9. The KA subsets the web and mobile knowledge units (often the most closely related units in CS Core) into foundations and specialized platforms core hours to provide flexibility and adaptability. This division allows programs at different institutions to offer different interests, concentrations, or degrees that focus on different application areas, where many of these concepts intersect the CS core. Therefore, the *common aspects web* and mobile foundations have concepts in many CS programs' core. Finally, the rest of the knowledge units permit the curriculum to have an extended and flexible number of KA core hours.
- **Renamed old knowledge units and incorporated new ones**: in the spirit of capturing the future technology and societal responsibilities, the Robotics, Embedded Systems, and Society, Ethics, and Professionalism (SEP) knowledge units were introduced in this version. These KU work harmoniously with other KAs, consistent with the topics and concepts covered in specific KAs' knowledge units.

Specialized platform development provides a deep understanding of a particular user group's needs and requirements and considers other knowledge areas that help design and build a platform that meets other KA's needs. Considering other KAs' needs, SPD helps to streamline workflows, improve efficiency, and drive innovation across the recommended curriculum discussed in CS2023.

Core Hours

Knowledge Units	**CS Core**	**KA Core**
SPD/Common Aspects	3	*
SPD/Web Platforms		*
SPD/Mobile Platforms		*
SPD/Robot Platforms		*
SPD/Embedded Platforms		*
SPD/Game Platforms		*
SPD/Interactive Computing Platforms		*
Total	**3**	

Knowledge Units

SPD-Common: Common Aspects/Shared Concerns

CS-Core:

1. Overview of development platforms (i.e., web, mobile, game, robotics, embedded, and Interactive)
    a. Input/sensors/control devices/haptic devices
    b. Resource constraints
        i. Computational
        ii. Data storage
        iii. Memory
        iv. Communication
    c. Requirements
        i. security, uptime availability, fault tolerance (See also: SE-Reliability)
    d. Output/actuators/haptic devices
2. Programming via platform-specific Application Programming Interface (API) vs traditional application construction
3. Overview of platform Languages, (e.g., Python, Swift, Lua, Kotlin)
4. Programming under platform constraints and requirements (e.g., available development tools, development, security considerations)
5. Techniques for learning and mastering a platform-specific programming language.

Illustrative Learning Outcomes

1. List the constraints of mobile programming
2. Describe the three-tier model of web programming
3. Describe how the state is maintained in web programming
4. List the characteristics of scripting languages

SPD-Web: Web Platforms

KA-Core:

1. Web programming languages (e.g., HTML5, JavaScript, PHP, CSS)
2. Web platforms, frameworks, or meta-frameworks
    a. Cloud services
    b. API, Web Components
3. Software as a Service (SaaS)
4. Web standards such as document object model, accessibility (See also: HCI-Accessability: 2)
5. Security and Privacy considerations (See also, SEP-Security)

Non-Core:

6. Analyzing requirements for web applications
7. Computing services (see also, DM-NoSQL: 8)
    a. Cloud Hosting
    b. Scalability (e.g., Autoscaling, Clusters)
    c. How to estimate costs for these services (based on requirements or time usage)
8. Data management (See also: DM-Core)
    a. Data residency (where the data is located and what paths can be taken to access it)

    b. Data integrity: guaranteeing data is accessible and guaranteeing that data is deleted when required
9. Architecture
    a. Monoliths vs. Microservices
    b. Micro-frontends
    c. Event-Driven vs. RESTful architectures: advantages and disadvantages
    d. Serverless, cloud computing on demand
10. Storage solutions (See also: DM-Relational/ DM-SQL)
    a. Relational Databases
    b. NoSQL databases

Illustrative Learning Outcomes

1. Design and Implement a web application using microservice architecture design.
2. Describe the web platform's constraints and opportunities, such as hosting, services, and scalability, that developers should consider.
3. Compare and contrast web programming with general-purpose programming.
4. Describe the differences between Software-as-a-Service and traditional software products.
5. Discuss how web standards impact software development.
6. Review an existing web application against a current web standard.

SPD-Mobile: Mobile Platforms

KA-Core:

1. Development
    a. Mobile programming languages
    b. Mobile programming environments
2. Mobile platform constraints
    a. User interface design (See also: HCI: Understanding the User, GIT-Image Processing)
    b. Security
3. Access
    a. Accessing data through APIs (See also: DM-Quering)
    b. Designing API endpoints for mobile apps: pitfalls and design considerations
    c. Network and the Web interfaces (See also: NC-Introduction, DM-Modeling:8.d)

Non-Core:

4. Development
    a. Native versus cross-platform development
    b. Software design/architecture patterns for mobile applications (See also: SE-Design)
5. Mobile platform constraints
    a. Responsive user interface design (see also: HCI-Accessibility and Inclusive Design)
    b. Diversity and mobility of devices
    c. User experiences differences (e.g., between mobile and web-based applications)
    d. Power and performance tradeoff
6. Mobile computing affordances

    a. Location-aware applications
    b. Sensor-driven computing (e.g., gyroscope, accelerometer, health data from a watch)
    c. Telephony and instant messaging
    d. Augmented reality (See also: GIT-Immersion)
7. Specification and testing (See also: SDF: Software Development Practices, SE-Validation)
8. Asynchronous computing (See also: PDC)
    a. Difference from traditional synchronous programming
    b. Handling success via callbacks
    c. Handling errors asynchronously
    d. Testing asynchronous code and typical problems in testing

Illustrative Learning Outcomes

1. Develop a location-aware mobile application with data API integration.
2. Build a sensor-driven mobile application capable of logging data on a remote server.
3. Create a communication appincorporating telephony and instant messaging.
4. Compare and contrast mobile programming with general-purpose programming.
5. Evaluate the pros and cons of native and cross-platform mobile app development.

SPD-Robot: Robot Platforms

Non-Core: (See also: AI-Robo)

1. Types of robotic platforms and devices
2. Sensors, embedded computation, and effectors (actuators) (See also: GIT-Physical:Tangible)
3. Robot-specific languages and libraries
4. Robotic platform constraints and design considerations
5. Interconnections with physical or simulated systems
6. Robotics
    a. Robotic software architecture (e.g., using the Robot Operating System)
    b. Forward kinematics (See also: GIT-Animation)
    c. Inverse kinematics (See also: GIT-Animation)
    d. Dynamics
    e. Navigation and robotic path planning (See also: AI-Robo)
    f. Manipulation and grasping (See also: AI-Robo)
    g. Safety considerations (See also: SEP-Professional, SEP-Context) // move to SEP

Illustrative Learning Outcomes

1. Design and implement an application on a given robotic platform
2. Assemble an Arduino-based robot kit and program it to navigate a maze
3. Compare robot-specific languages and techniques with those used for general-purpose software development
4. Explain the rationale behind the design of the robotic platform and its interconnections with physical or simulated systems,
5. Given a high-level application, design a robot software architecture using ROS specifying all components and interconnections (ROS topics) to accomplish that application
6. Discuss the constraints a given robotic platform imposes on developers

SPD-Embedded: Embedded Platforms

This Knowledge unit considers embedded computing platforms and their applications. Embedded platforms cover knowledge ranging from sensor technology to ubiquitous computing applications.

Non-Core:

1. Introduction to the unique characteristics of embedded systems
    a. real-time vs. soft real-time and non-real-time systems
    b. Resource constraints, such as memory profiles, deadlines (See also: AR-Memory: 2)
2. API for custom architectures
    a. GPU technology (See also: AR-Heterogeneity: 1, GIT-Interfaces (AI))
    b. Field Programmable Gate Arrays (See also: AR-Logic: 2)
    c. Cross platform systems
3. Embedded Systems
    a. Microcontrollers
    b. Interrupts and feedback
    c. Interrupt handlers in high-level languages (See also: SF-Overview: 5)
    d. Hard and soft interrupts and trap-exits (See also: OS-Principles: 6)
    e. Interacting with hardware, actuators, and sensors
    f. Energy efficiency
    g. Loosely timed coding and synchronization
    h. Software adapters
4. Real-time systems
    a. Hard real-time systems vs. soft real-time systems (See also: OS-Real-time: 3)
    b. Timeliness
    c. Time synchronization/scheduling
    d. Prioritization
    e. Latency
    f. Compute jitter
5. Memory management
    a. Mapping programming construct (variable) to a memory location(See also: AR-Memory)
    b. Shared memory (See also: OS-Memory)
    c. Manual memory management
    d. Garbage collection (See also: FPL-Language Translation)
6. Safety considerations and safety analysis (See also: SEP-Context, SEP-Professional)
7. Sensors and actuators
8. Embedded programming
9. Real-time resource management
10. Analysis and verification
11. Application design

Illustrative Learning Outcomes

1. Design and implement a small embedded system for a given platform (e.g., a smart alarm clock or a drone)
2. Describe the unique characteristics of embedded systems versus other systems

3. Interface with sensors/actuators
4. Debug a problem with an existing embedded platform
5. Identify different types of embedded architectures
6. Evaluate which architecture is best for a given set of requirements
7. Design and develop software to interact with and control hardware
8. Design methods for real-time systems
9. Evaluate real-time scheduling and schedulability analysis
10. Evaluate formal specification and verification of timing constraints and properties

SPD-Game: Game Platforms

The Game Platforms knowledge unit draws attention to concepts related to the engineering of performant real-time interactive software on constrained computing platforms. Material on requirements, design thinking, quality assurance, and compliance enhances problem-solving skills and creativity.

Non-Core:

1. Historical and contemporary platforms for games (See also: AR-A: Digital Logic and Digital Systems, (See also: AR-A: Interfacing and Communication)
    a. *Evolution of Game Platforms*
    (e.g., Brown Box to Metaverse and beyond; Improvement in Computing Architectures (CPU and GPU); Platform Convergence and Mobility)
    b. *Typical Game Platforms*
    (e.g., Personal Computer; Home Console; Handheld Console; Arcade Machine; Interactive Television; Mobile Phone; Tablet; Integrated Head-Mounted Display; Immersive Installations and Simulators; Internet of Things enabled Devices; CAVE Systems; Web Browsers; Cloud-based Streaming Systems)
    c. *Characteristics and Constraints of Different Game Platforms*
    (e.g., Features (local storage, internetworking, peripherals); Run-time performance (GPU/CPU frequency, number of cores); Chipsets (physics processing units, vector co-processors); Expansion Bandwidth (PCIe); Network throughput (Ethernet); Memory types and capacities (DDR/GDDR); Maximum stack depth; Power consumption; Thermal design; Endian)
    d. *Typical Sensors, Controllers, and Actuators* (See also: GIT-Interaction)
    (e.g., typical control system designs—peripherals (mouse, keypad, joystick), game controllers, wearables, interactive surfaces; electronics and bespoke hardware; computer vision, inside-out tracking, and outside-in tracking; IoT-enabled electronics and i/o.

   e. *Esports Ecosystems*
      (e.g., evolution of gameplay across platforms; games and esports; game events such as LAN/arcade tournaments and international events such the Olympic Esports Series; streamed media and spectatorship; multimedia technologies and broadcast management; professional play; data and machine learning for coaching and training)
2. Real-time Simulation and Rendering Systems
   a. CPU and GPU architectures: (See also, AR-Heteroginity: 2)
      (e.g., Flynn's taxonomy; parallelization; instruction sets; common components—graphics compute array, graphics memory controller, video graphics array basic input/output system; bus interface; power management unit; video processing unit; display interface)
   b. Pipelines for physical simulations and graphical rendering: (See also, GIT-Rendering)
      (e.g., tile-based, immediate-mode)
   c. *Common Contexts for Algorithms, Data Structures, and Mathematical Functions*: (See also, MSF:?, AL-Fundamentals)
      (e.g., game loops; spatial partitioning, viewport culling, and level of detail; collision detection and resolution; physical simulation; behavior for intelligent agents; procedural content generation)
   d. Media representations (See also, GIT-Fundamentals: 8, GIT-Geometric: 3)
      (e.g., i/o, and computation techniques for virtual worlds: audio; music; sprites; models and textures; text; dialogue; multimedia (e.g., olfaction, tactile).
3. Game Development Tools and Techniques
   a. *Programming Languages*:
      (e.g., C++; C#; Lua; Python; JavaScript)
   b. Shader Languages: HLSL; GLSL; ShaderGraph.
   c. *Graphics Libraries and APIs* (See also, GIT-Rendering, HCI-System Design: 3)
      (e.g., DirectX; SDL; OpenGL; Metal; Vulkan; WebGL)
   d. *Common Development Tools and Environments*: (See also: SDF-Practices: 2, SE-Tools,)
      (e.g., IDEs; Debuggers; Profilers; Version Control Systems including those handling binary assets; Development Kits and Production/Consumer Kits; Emulators)
4. Game Engines
   a. Open Game Engines
      (e.g.,Unreal; Unity; Godot; CryEngine; Phyre; Source 2; Pygame and Ren'Py; Phaser; Twine; SpringRTS)
   b. *Techniques (See also: AR-Performance and Energy Efficiency, (See also: SE-Requirements)*
      (e.g., Ideation; Prototyping; Iterative Design and Implementation; Compiling Executable Builds; Development Operations and Quality Assurance—Play Testing and Technical Testing; Profiling; Optimization; Porting; Internationalization and Localization; Networking)
5. Game Design
   a. Vocabulary
      (e.g., game definitions; mechanics-dynamics-aesthetics model; industry terminology; experience design; models of experience and emotion)

   b. *Design Thinking and User-Centered Experience Design* (See also: SE-?)
   (e.g., methods of designing games; iteration, incrementing, and the double-diamond; phases of pre- and post-production; quality assurance, including alpha and beta testing; stakeholder and customer involvement; community management)
   c. *Genres*
   (e.g., adventure; walking simulator; first-person shooter; real-time strategy; multiplayer online battle arena (MOBA); role-playing game (rpg))
   d. *Audiences and Player Taxonomies (See also: HCI-Understanding the User: 5)*
   (e.g., people who play games; diversity and broadening participation; pleasures, player types, and preferences; Bartle, yee)
   e. *Proliferation of digital game technologies to domains beyond entertainment*: (See also: AI-E: Applications and Societal Impact)
   (e.g., Education and Training; Serious Games; Virtual Production; Esports; Gamification; Immersive Experience Design; Creative Industry Practice; Artistic Practice; Procedural Rhetoric.)

Illustrative Learning Outcomes

1. Recall the characteristics of common general-purpose graphics processing architectures
2. Identify the key stages of the immediate-mode rendering pipeline
3. Describe the key constraints a given game platform will likely impose on developers
4. Explain how esports are streamed to large audiences over the Internet
5. Translate complex mathematical functions into performant source code
6. Use an industry-standard graphics API to render a 3D model in a virtual scene
7. Modify a shader to change a visual effect according to stated requirements
8. Implement a game for a particular platform according to specification
9. Optimize a function for processing collision detection in a simulated environment
10. Assess a game's run-time and memory performance using an industry-standard tool and development environment
11. Compare the interfaces of different game platforms, highlighting their respective implications for human-computer interaction
12. Recommend an appropriate set of development tools and techniques for implementing a game of a particular genre for a given platform
13. Discuss the key challenges in making a digital game that is cross-platform compatible
14. Suggest how game developers can enhance the accessibility of a game interface
15. Create novel forms of gameplay using frontier game platforms

SPD-Interactive: Interactive Computing Platforms

Non-Core:

1. Data Analysis Platforms
   a. Jupyter notebooks; Google Colab; R; SPSS; Observable
   b. Cloud SQL/data analysis platforms (e.g., BigQuery)(See also: DM-Quering)
      i. Apache Spark
2. Data Visualizations (See also: GIT:Visualization)
   a. Interactive presentations backed by data

    b. Design tools requiring low-latency feedback loops
        i. rendering tools
        ii. graphic design tools
3. Programming by Prompt
    a. Generative AI (e.g., OpenAI's ChatGPT, OpenAI's Codex, GitHub's Copilot) and LLMs are accessed/interacted.
    b. Quantum Platforms (See also: AR-Quantum, FPL-Quantum: Quantum Computing)
        i. Program quantum logic operators in quantum machines
        ii. Use API for available quantum services
        iii. Signal analysis / Fourier analysis / Signal processing (for music composition, audio/RF analysis) (See also: GIT-Image)

Illustrative Learning Outcomes

1. Interactively analyze large datasets
2. Create a backing track for a musical performance (e.g., with live coding)
3. Create compelling computational notebooks that construct a narrative for a given journalistic goal/story.
4. Implement interactive code that uses a dataset and generates exploratory graphics
5. Create a program that performs a task using LLM systems
6. Contrast a program developed by an AI platform and by a human
7. Implement a system that interacts with a human without using a screen
8. Contextualize the attributes of different data analysis styles, such as interactive vs. engineered pipeline
9. Write a program using a notebook computing platform (e.g., searching, sorting, or graph manipulation)
10. Demonstrate a quantum gate outcome using a quantum platform

SPD-SEP/Mobile

Topics

1. Privacy and data protection
2. Accessibility in mobile design
3. Security and cybersecurity:
4. Social impacts of mobile technology
5. Ethical use of AI and algorithms

Illustrative Learning Outcomes

1. Understand and uphold ethical responsibilities for safeguarding user privacy and data protection in mobile applications.
2. Design mobile applications with accessibility in mind, ensuring effective use by people with disabilities.

3. Demonstrate proficiency in secure coding practices to mitigate risks associated with various security threats in mobile development.
4. Analyze the broader social impacts of mobile technology, including its influence on communication patterns, relationships, and mental health.
5. Comprehend the ethical considerations related to the use of AI in mobile applications, ensuring algorithms are unbiased and fair.

SPD-SEP/Web

Topics

1.

Illustrative Learning Outcomes

1. Understand how mobile computing impacts communications and the flow of information within society
2. Design mobile apps that have made daily tasks easier/faster
3. Recognize how the ubiquity of mobile computing has affected work-life balance
4. Understand how mobile computing impacts health monitoring and healthcare services
5. Comprehend how mobile apps are used to educate on and help achieve UN sustainability goals

SPD-SEP/Game

Topics

1. Intellectual Property Rights in Creative Industries:
    a. *Intellectual Property Ownership*: copyright; trademark; design right; patent; trade secret; civil versus criminal law; international agreements; procedural content generation and the implications of generative artificial intelligence.
    b. *Licensing*: usage and fair usage exceptions; open-source license agreements; proprietary and bespoke licensing; enforcement.
2. Fair Access to Play:
    a. *Game Interface Usability*: user requirements; affordances; ergonomic design; user research; experience measurement; heuristic evaluation methods for games.
    b. *Game Interface Accessibility:* forms of impairment and disability; means to facilitate access to games; universal design; legislated requirements for game platforms; compliance evaluation; challenging game mechanics and access.
3. Game-Related Health and Safety:
    a. *Injuries in Play:* ways of mitigating common upper body injuries, such as repetitive strain injury; exercise psychology and physiotherapy in esports.
    b. *Risk Assessment for Events and Manufacturing:* control of substances hazardous to health (COSHH); fire safety; electrical and electronics safety; risk assessment for games and game events; risk assessment for manufacturing.
    c. *Mental Health*: motivation to play; gamification and gameful design; game psychology—internet gaming disorder.
4. Platform Hardware Supply Chain and Sustainability:
    a. *Platform Lifecycle*: platform composition—materials, assembly; mineral excavation and processing; power usage; recycling; planned obsolescence.

   b. *Modern Slavery*: supply-chains; forced labour and civil rights; working conditions; detection and remission; certification bodies and charitiable endeavours.
5. *Representation in the Media and Industry:*
   a. *Inclusion*: identity and identification; exclusion of characters diverse audiences identify with; media representation and its effects; media literacy; content analysis; stereotyping; sexualization.
   b. *Equality*: histories and controversies, such as gamergate; quality of life in the industry; professional discourse and conduct in business contexts; pathways to game development careers; social mobility; experience of developers from different backgrounds and identities; gender, and technology.

Illustrative Learning Outcomes

1. Recall the ways in which creators can protect their intellectual property
2. Identify common pitfalls in game interfaces that exclude players with impaired or non-functional vision
3. Describe how heuristic evaluation can be used to identify usability problems in game interfaces
4. Explain why upper body injuries are common in esports
5. Reform characters and dialogues in a scene to reduce stereotype threat
6. Illustrate the way in which the portrayal of race in a game can influence the risk of social exclusion in the associated online community around the game
7. Modify a policy for a LAN party event to include mitigations that lower the risk of fire
8. Design a gamification strategy to motivate serious play for an awareness-raising game
9. Analyse the role of company hiring policies and advocacy on social mobility
10. Assess the appropriateness of two manufacturers for producing a new games console
11. Compare options for open-source licensing of a game development tool
12. Recommend changes to a game interface to improve access to players who are deaf or whose hearing is otherwise impaired
13. Discuss whether games are addictive in nature
14. Suggest how the portrayal of women in video games influences the way players may perceive members of those groups
15. Create a videogame that successfully advocates for climate science

SPD-SEP/Robotics

SPD-SEP/Interactive

This knowledge unit captures the society, ethics, and professionalism aspects from the specialized platform development viewpoint. Every stage from the software development perspective impacts the SEP knowledge unit.

Topics

1. Augmented technology and societal impact
2. Robotic design

3. Graphical User Interfaces Considerations for DEI
4. Recognizing data privacy and implications
5. LLMs and global compliance regulations, such as copyright law
6. Mobile development and global equality

Professional Dispositions

- Learning to learn (new platforms, languages)
- Inventiveness (in designing software architecture within non-traditional constraints)
- Adaptability (to new constraints)

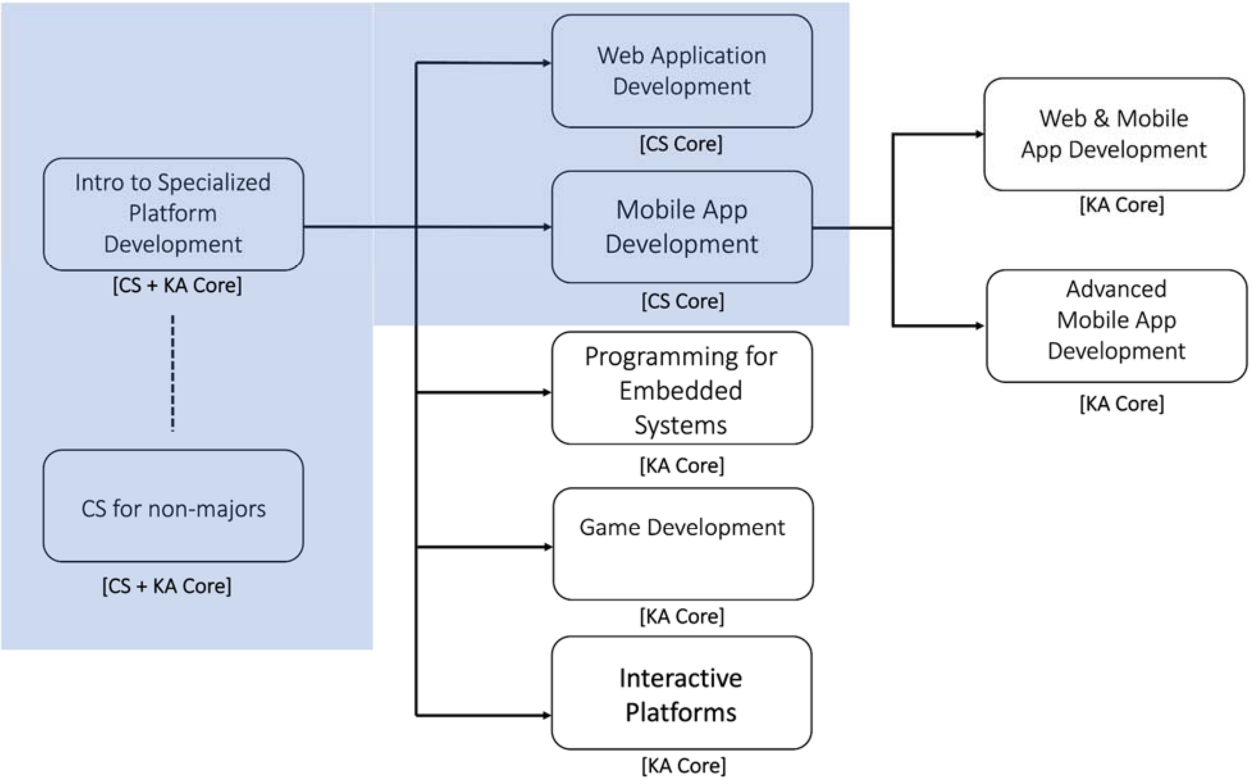
Math Requirements

Required:

Desired:

- Equations and Basic Algebra
- Geometry (e.g., 2d and 3d coordinate systems—cartesian and polar—points, lines, and angles)
- Trigonometry
- Vectors, Matrices, and Quaternions—linear transformations, affine transformations, perspective projections, exponential maps, rotation, etc.
- Geometric primitives
- Rigid body kinematics and Newtonian physics
- Signal processing
- Coordinate-space transformations
- Parametric curves
- Binary and Hexadecimal Number Systems
- Logic and Boolean Algebra
- Calculus
- Linear Algebra
- Probability/Statistics (e.g., dynamic systems, visualization e.g., algorithmically generated Tuftian-style displays)
- Discrete Math/Structures (e.g., graphs for process control and path search)

Course Packaging Suggestions



Courses Common to CS and KA Core Approaches
Specialized Platform Development

- **SPD-Common**
- **SPD-Web**
- **SPD-Mobile**
- **SPD-Robotic**
- **SPD-Interactive**

CS for Non-Majors

- **SPD-Common**
-

Mobile Development 8 Week Course

- SPD-Mobile
  - API Design and Development
  - User-Centered Design and the Mobile Platform
  - Software Engineering Applications in Mobile Computing (covers design patterns, testing, async programming, and the like in a mobile context)
  - Challenges with Mobile Computing Security
  - Societal Impact of Mobile Computing
  - Mobile Computing Capstone Course

Committee

Chair: Christian Servin (El Paso Community College, El Paso, TX, USA)

Members:

- Sherif G. Aly, The American University in Cairo, Egypt
- Yoonsik Cheon, The University of Texas at El Paso, El Paso, Texas, USA
- Eric Eaton, University of Pennsylvania, Philadelphia, PA, USA
- Claudia L. Guevara, Jochen Schweizer mydays Holding GmbH, Munich, Germany
- Larry Heimann, Carnegie Mellon University, Pittsburgh, Pennsylvania, USA
- Amruth N. Kumar, Ramapo College of New Jersey, Mahwah, NJ, USA
- R. Tyler Pirtle, Google
- Michael James Scott, Falmouth University, UK

Contributors:

- Sean R. Piotrowski, Rider University, USA
- Mark 0’Neil, Blackboard Inc., USA
- John DiGennaro, Qwickly
- Rory K. Summerley, London South Bank University, UK.

Core Topics and Hours

Artificial Intelligence (AI)

KA	KU	Topic	Skill	Core	Hours
AI	Fundamental Issues	● Overview of AI problems, Examples of successful recent AI applications ● Definitions of agents with examples (e.g., reactive, deliberative) ● What is intelligent behavior? ○ The Turing test and its flaws ○ Multimodal input and output ○ Simulation of intelligent behavior ○ Rational versus non-rational reasoning	Explain	CS	2
		4. Problem characteristics a. Fully versus partially observable b. Single versus multi-agent c. Deterministic versus stochastic d. Static versus dynamic e. Discrete versus continuous 5. Nature of agents a. Autonomous, semi-autonomous, mixed-initiative autonomy b. Reflexive, goal-based, and utility-based c. Decision making under uncertainty and with incomplete information d. The importance of perception and environmental interactions e. Learning-based agents f. Embodied agents i. sensors, dynamics, effectors	Evaluate		
		6. AI Applications, growth, and Impact (economic, societal, ethics)	Explain		
AI	Fundamental Issues	7. Additional depth on problem characteristics with examples 8. Additional depth on nature of agents with examples	Evaluate	KA	1

		9. Additional depth on AI Applications, growth, and Impact (economic, societal, ethics)			
AI AL *	Search Fundamental Data Structures and Algorithms *	● State space representation of a problem ○ Specifying states, goals, and operators ○ Factoring states into representations (hypothesis spaces) ○ Problem solving by graph search ■ e.g., Graphs as a space, and tree traversals as exploration of that space ■ Dynamic construction of the graph (you're not given it upfront)	Explain	CS	5
		● Uninformed graph search for problem solving ○ Breadth-first search ○ Depth-first search ■ With iterative deepening ○ Uniform cost search	Develop/Apply		
		● Heuristic graph search for problem solving ○ Heuristic construction and admissibility ○ Hill-climbing ○ Local minima and the search landscape ■ Local vs global solutions ○ Greedy best-first search ○ A* search	Develop/Apply		
		9. Space and time complexities of graph search algorithms	Evaluate		
AI	Search	● Bidirectional search ● Beam search ● Two-player adversarial games ○ Minimax search ○ Alpha-beta pruning ■ Ply cutoff ● Implementation of A* search	Develop/Apply	KA	3

AI	Fundamental Knowledge Representation and Reasoning	7. Types of representations a. Symbolic, logical i. Creating a representation from a natural language problem statement b. Learned subsymbolic representations c. Graphical models (e.g., naive Bayes, Bayes net) 8. Review of probabilistic reasoning, Bayes theorem (cross-reference with DS/Discrete Probability)	Explain	CS	2
		9. Bayesian reasoning a. Bayesian inference	Apply		
AI	Fundamental Knowledge Representation and Reasoning	• Random variables and probability distributions • Axioms of probability • Probabilistic inference • Bayes' Rule (derivation) • Bayesian inference (more complex examples) • Independence • Conditional Independence • Utility and decision making	Apply	KA	2
AI	Machine Learning	1. Definition and examples of a broad variety of machine learning tasks a. Supervised learning i. Classification ii. Regression b. Reinforcement learning c. Unsupervised learning i. Clustering 2. Fundamental ideas: a. no free lunch b. undecidability c. sources of error in machine learning		CS	4

		3. Simple statistical-based supervised learning such as Naive Bayes, Decision trees a. Focus on how they work without going into mathematical or optimization details; enough to understand and use existing implementations correctly 4. The overfitting problem and controlling solution complexity (regularization, pruning – intuition only) a. The bias (underfitting) - variance (overfitting) tradeoff 5. Working with Data a. Data preprocessing i. Importance and pitfalls of b. Handling missing values (imputing, flag-as-missing) i. Implications of imputing vs flag-as-missing c. Encoding categorical variables, encoding real-valued data d. Normalization/standardization e. Emphasis on real data, not textbook examples 6. Representations a. Hypothesis spaces and complexity b. Simple basis feature expansion, such as squaring univariate features c. Learned feature representations 7. Machine learning evaluation a. Separation of train, validation, and test sets b. Performance metrics for classifiers c. Estimation of test performance on held-out data d. Tuning the parameters of a machine learning model with a validation set e. Importance of understanding what your model is actually doing, where its pitfalls/shortcomings are, and the implications of its decisions 8. Basic neural networks	Apply/Develop/Evaluate		

		a. Fundamentals of understanding how neural networks work and their training process, without details of the calculations			
		9. Ethics for Machine Learning a. Focus on real data, real scenarios, and case studies. b. Dataset/algorithmic/evaluation bias	Explain / Evaluate		
AI	Machine Learning	● Formulation of simple machine learning as an optimization problem, such as least squares linear regression or logistic regression ○ Objective function ○ Gradient descent ○ Regularization to avoid overfitting (mathematical formulation) ● Ensembles of models ○ Simple weighted majority combination ● Deep learning ○ Deep feed-forward networks (intuition only, no math) ○ Convolutional neural networks (intuition only, no math) ○ Visualization of learned feature representations from deep nets ● Performance evaluation ○ Other metrics for classification (e.g., error, precision, recall) ○ Performance metrics for regressors ○ Confusion matrix ○ Cross-validation	Apply/ Develop/ Evaluate	KA	4

		■ Parameter tuning (grid/random search, via cross-validation) ● Overview of reinforcement learning ● Two or more applications of machine learning algorithms ○ E.g., medicine and health, economics, vision, natural language, robotics, game play			
		● Ethics for Machine Learning ○ Continued focus on real data, real scenarios, and case studies. ○ Privacy ○ Fairness	Explain/ Evaluate		
AI	Applications and Societal Impact	● *Note: For the CS core, cover at least one application and an overview of the societal issues of AI/ML.* ● ● Applications of AI to a broad set of problems and diverse fields, such as medicine, health, sustainability, social media, economics, education, robotics, etc. (choose one for CS Core, at least one additional for KA core) ○ Formulating and evaluating a specific application as an AI problem ○ Data availability and cleanliness ■ Basic data cleaning and preprocessing ■ Data set bias ○ Algorithmic bias ○ Evaluation bias ● Deployed deep generative models ○ High-level overview of deep image generative models (e.g. as of 2023, DALL-E, Midjourney, Stable	Explain/ Evaluate	CS	2

		Diffusion, etc.), how they work, their uses, and their shortcomings/pitfalls. ○ High-level overview of large language models (e.g. as of 2023, ChatGPT, Bard, etc.), how they work, their uses, and their shortcomings/pitfalls. ● Societal impact of AI ○ Ethics ○ Fairness ○ Trust / explainability			
AI	Applications and Societal Impact	● *Note: The KA core should go more in-depth with one or more additional applications, more in-depth on deep generative models, and an analysis and discussion of the social issues.* ● ● Applications of AI to a broad set of problems and diverse fields, such as medicine, health, sustainability, social media, economics, education, robotics, etc. (choose one for CS Core, at least one additional for KA core) ○ Formulating and evaluating a specific application as an AI problem ○ Data availability and cleanliness ■ Basic data cleaning and preprocessing ■ Data set bias ○ Algorithmic bias ○ Evaluation bias ● Deployed deep generative models ○ High-level overview of deep image generative models (e.g. as of 2023, DALL-E, Midjourney, Stable Diffusion, etc.), how they work, their uses, and their shortcomings/pitfalls. ○ High-level overview of large language models (e.g. as of 2023, ChatGPT, Bard, etc.), how they work, their uses, and their shortcomings/pitfalls. ● Societal impact of AI	Explain/Evaluate	KA	2

		○ Ethics ○ Fairness ○ Trust / explainability			

Algorithmic Foundations (AL)					
KA	**KU**	**Topic**	**Performance**	**CS/KA**	**Hours**
AL	Foundational Complexity	2. Arrays (single & multi dimension, strings) 1a. ADT and Dictionary operations 2b i. Foundational complexity classes: Constant O(1)	Explain Apply Explain	CS	1
AL	Foundational Complexity Strategies	11a. Search O(n), (e.g., Linear search of an array) 2b iii. Foundational complexity classes: Linear O(n) 1a. Brute Force	Apply Evaluate Explain	CS	0.5
AL	Foundational Complexity Strategies	12a. Sorting O(n^2), (e.g., Selection sort of an array) 2b v. Foundational complexity classes: Quadratic O(n^2) 1a. Brute Force	Apply Evaluate Explain	CS	0.5
AL	Foundational Complexity Strategies	11b. Search O($\log_2 n$), (e.g., Binary search of an array) 2b ii. Foundational complexity classes: Logarithmic 1b ii. Decrease and Conquer	Apply Evaluate Explain		1
AL	Foundational Complexity Strategies	12b. Sorting O($n \log n$), (e.g., Quick, Merge, Tim: array) 2b iv. Foundational complexity classes: Log Linear 1c. Divide-and-Conquer	Apply Evaluate Explain		1
AL	Foundational Complexity Strategies	4. Linked Lists 1a. Dictionary Operations 11a. Search O(n), (e.g., Linear search of a linked list) 2b iii. Foundational complexity classes: Linear O(n) 1a. Brute Force	Explain Apply Apply Evaluate Explain		1
AL	Foundational Complexity	5. Stacks 1a. Dictionary Operations (push, pop) 2b iii. Foundational complexity classes: Constant O(n) 6. Queues and Deques	Explain Apply Explain		1
	Foundational	7. Hash tables / Maps 1a. Dictionary Operations (put, get) 7a. Collision resolution and complexity	Explain Apply Explain		1

	Complexity Strategies	2b iii. Foundational complexity classes: Constant O(n) 1f. Time vs. Space tradeoff	Explain Explain		1
	Foundational” Strategies Foundational Strategies	9. Trees 1a. Dictionary operations (insert, delete) 11c. Search DFS/BFS 2b. Decrease and Conquer 9b. Balanced Trees (e.g., AVL, 2-3, Red-Black, Heaps) 1e ii. Transform and Conquer: Representation change	Explain Apply Apply Explain Apply Explain		1 2
	Foundational Foundational Strategies Foundational Strategies	8. Graphs (e.g., [un]directed, [a]cyclic, [un]connected, [un]weighted) 8a. Representation: adjacency list vs. matrix 13. Graph Algorithms 13a. Shortest Path (e.g., Dijkstra’s, Floyd’s) 1d. Greedy 1e iv. Dynamic Programming 13b. Minimal, spanning tree (e.g. Prim’s, Kruskal’s) 1d. Greedy	Explain		3
	Foundational	1. Abstract Data Types 3. Records/Structures/Tuples 10. Sets			1
AL	Strategies Strategies Strategies Strategies	1. Paradigms demonstrated in AL-Fundamentals algorithms: Brute-Force, Decrease-and-Conquer, Divide-and-Conquer, Iteration vs. Recursion, Time-Space Tradeoff, 1e. Transform-and-Conquer 1e i. Instance simplification (Find duplicates by pre-sorting) 1e iii. Problem reduction (Least-common-multiple) 2. Handling Exponential Growth e.g., A*, Backtracking, Branch-and-Bound 1e iv. Dynamic Programming e.g., Bellman-Ford, Knapsack, Floyd-Warshall	Explain Explain Explain		3 1 1 1
AL	Complexity Complexity	1. Analysis Framework 2. Asymptotic complexity analysis Big O, Little O, Big Omega, and Big Theta 2b. Foundational complexity classes demonstrated by AL-Fundamentals algorithms: Constant, Logarithmic, Linear, Log Linear, Quadratic, and Cubic	Explain	CS	1 1

	Complexity	4. Tractability and Intractability Foundational Complexity Classes: Exponential O(2^n) P, NP and NP-C complexity classes Reductions Problems Traveling Salesperson, Knapsack, SAT			4
	Strategies	1. Paradigms: Exhaustive brute force, Dynamic Programming			
	Complexity	2b viii. Factorial complexity classes: Factorial O(n!) All Permutations, Hamiltonian Circuit			
AL	Models	1a. Finite State Automata 2a. Regular language, grammar, and expressions	Apply Explain	CS	1
	Models	1b. Pushdown Automata 2b. Context-Free language and grammar	Apply Explain		1
	Models	1d. Turing Machine 2d. Recursively Enumerable language and grammar 1c. Linear-Bounded 2c. Context-Sensitive language and grammar	Explain		2
	Models	4. Decidability, Computability, Halting problem 5. The Church-Turing Thesis			1
	Models	6. Algorithmic Correctness Invariants (e.g., in: iteration, recursion, tree search)			1
AL	SEP	1. Social, Ethical, and Secure Algorithms 2. Algorithmic Fairness (e.g. differential privacy) 3. Accountability/Transparency 4. Responsible algorithms 5. Economic and other impacts of algorithms 6. Sustainability	Explain	CS	In SEP Hours

Architecture and Organization (AR)

KA	KU	Topic	Skill	Core	Hours
AR	Digital Logic and Digital Systems	● Overview and history of computer architecture ● Combinational vs. sequential logic ○ Fundamental combinational ○ Sequential logic building block ● Functional hardware and software multi-layer architecture	Explain	KA	3
	Digital Logic and Digital Systems	● Computer-aided design tools that process hardware and architectural representations	Evaluate		

		● High-level synthesis ○ Register transfer notation ○ Hardware description language (e.g. Verilog/VHDL/Chisel) ● System-on-chip (SoC) design flow ● Physical constraints ○ Gate delays ○ Fan-in and fan-out ○ Energy/power [Shared with SPD] ○ Speed of light			
AR	Machine-Level Data Representation	● Bits, bytes, and words ● Numeric data representation and number bases ○ Fixed-point ○ Floating-point ● Signed and twos-complement representations ● Representation of non-numeric data ● Representation of records and arrays	Apply	CS	1
AR	Assembly Level Machine Organization	● von Neumann machine architecture ● Control unit; instruction fetch, decode, and execution ● Introduction to SIMD vs. MIMD and the Flynn taxonomy ● Shared memory multiprocessors/multicore organization [Shared with OS]	Explain	CS	1
AR	Assembly Level Machine Organization	● Instruction set architecture (ISA) (e.g. x86, ARM and RISC-V) ○ Instruction formats ○ Data manipulation, control, I/O ○ Addressing modes ○ Machine language programming ○ Assembly language programming ● Subroutine call and return mechanisms (xref PL/language translation and execution) [Shared with OS] ● I/O and interrupts [Shared with OS] ● Heap, static, stack and code segments [Shared with OS]	Develop	KA	2
		● Memory hierarchy: the importance of temporal and spatial locality [Shared with OS]	Explain		

AR	Memory Hierarchy	● Main memory organization and operations ● Persistent memory (e.g. SSD, standard disks) [Shared with OS] ● Latency, cycle time, bandwidth and interleaving ● Virtual memory (hardware support, cross-reference OS/Virtual Memory) [Shared with OS] ● Fault handling and reliability [Shared with OS] ● Reliability (cross-reference SF/Reliability through Redundancy) ○ Error coding ○ Data compression ○ Data integrity ● Non-von Neumann Architectures ○ In-Memory Processing (PIM)		CS	6
		● Cache memories [Shared with OS] ○ Address mapping ○ Block size ○ Replacement and store policy ● Multiprocessor cache coherence	Evaluate		
AR	Interfacing and Communication	● I/O fundamentals[Shared with OS and SPD] ○ Handshaking and buffering ○ Programmed I/O ○ Interrupt-driven I/O ● Interrupt structures: vectored and prioritized, interrupt acknowledgement [Shared with OS] ● External storage, physical organization and drives [Shared with OS] ● Buses fundamentals [Shared with OS and SPD] ○ Bus protocols ○ Arbitration ○ Direct-memory access (DMA) ● Network-on-chip (NoC)	Explain	CS	1
AR	Functional Organization	● Implementation of simple datapaths, including instruction pipelining, hazard detection and resolution ● Control unit ○ Hardwired implementation ○ Microprogrammed realization	Develop	KA	2
		● Instruction pipelining	Explain		

		● Introduction to instruction-level parallelism (ILP)			
AR	Performance and Energy Efficiency	● Performance-energy evaluation (introduction): performance, power consumption, memory and communication costs ● Branch prediction, speculative execution, out-of-order execution, Tomasulo's algorithm ● Prefetching	Evaluate	KA	2
AR	Performance and Energy Efficiency	● Enhancements for vector processors and GPUs ● Hardware support for Multithreading ○ Race conditions ○ Lock implementations ○ Point-to-point synchronization ○ Barrier implementation ● Scalability ● Alternative architectures, such as VLIW/EPIC, accelerators and other special-purpose processors ● Dynamic voltage and frequency scaling (DVFS) ● Dark Silicon	Explain	KA	1
AR	Heterogeneous Architectures	● SIMD and MIMD architectures (e.g. General-Purpose GPUs, TPUs and NPUs) ● Heterogeneous memory system ○ Shared memory versus distributed memory ○ Volatile vs non-volatile memory ○ Coherence protocols ● Domain-Specific Architectures (DSAs) ○ Machine Learning Accelerator ○ In-networking computing ○ Embedded systems for emerging applications ○ Neuromorphic computing ● Packaging and integration solutions such as 3DIC and Chiplets	Explain	KA	3
AR	Quantum Architectures	● Principles ● The wave-particle duality principle ● The uncertainty principle in the double-slit experiment	Explain	KA	3

		● What is a Qubit? Superposition and measurement. Photons as qubits. ● Systems of two qubits. Entanglement. Bell states. The No-Signaling theorem. ● Axioms of QM: superposition principle, measurement axiom, unitary evolution ● Single qubit gates for the circuit model of quantum computation: X, Z, H. ● Two qubit gates and tensor products. Working with matrices. ● The No-Cloning Theorem. The Quantum Teleportation protocol. ● Algorithms ● Simple quantum algorithms: Bernstein-Vazirani, Simon's algorithm. ● Implementing Deutsch-Josza with Mach-Zehnder Interferometers. ● Quantum factoring (Shor's Algorithm) ● Quantum search (Grover's Algorithm) ● Implementation aspects ● The physical implementation of qubits (there are currently nine qubit modalities) ● Classical control of a Quantum Processing Unit (QPU) ● Error mitigation and control. NISQ and beyond. ● Emerging Applications ● Post-quantum encryption ● The Quantum Internet ● Adiabatic quantum computation (AQC) and quantum annealing			

Data Management (DM)					
KA	**KU**	**Topic**	**Skill**	**Core**	**Hours**
DM	The Role of Data	● The Data Life Cycle ● The social/legal aspects of data collections: [crosslist SEP]	Evaluate	CS	2

DM	Core DB System Concepts	● Purpose and advantages of database systems ● Components of database systems ● Design of core DBMS functions (e.g., query mechanisms, transaction management, buffer management, access methods) ● Database architecture, data independence, and data abstraction ● Transaction mgmt ● Normalization ● Approaches for managing large volumes of data (e.g., noSQL database systems, use of MapReduce). [crosslist PD] ● Distributed databases/cloud-based systems ● Structured, semi-structured, and unstructured databases	Explain	CS	2
		● Use of a declarative query language	Develop		
DM	Core DB System Concepts	● Systems supporting structured and/or stream content	Explain	KA	.5
DM	Data Modeling	● Data modeling ● Relational data models	Develop	CS	2
DM	Data Modeling	● Conceptual models (e.g., entity-relationship, UML diagrams) ● Semi-structured data model (expressed using DTD, XML, or JSON Schema, for example)	Explain	KA	3
DM	Relational Databases	● Entity and referential integrity ○ Candidate key, superkeys ● Relational database design	Explain	CS	1
DM	Relational Databases	● Mapping conceptual schema to a relational schema ● Physical database design: file and storage structures ● Functional dependency Theory ● Normalization Theory	Develop	KA	3

		○ Decomposition of a schema; lossless-join and dependency-preservation properties of a decomposition Normal forms (BCNF) ○ Denormalization (for efficiency)			
DM	Query Construction	● SQL Query Formulation	Develop	CS	2
DM	Query Construction	● Relational Algebra ● SQL ○ Data definition including integrity and other constraints specification ○ Update sublanguage	Develop	KA	4
DM	Query Processing	● Index structures ○ B+ trees ○ Hash indices: static and dynamic ○ Index creation in SQL ● Algorithms for query operators ○ External Sorting ○ Selection ○ Projection; with and without duplicate elimination ○ Natural Joins: Nested loop, Sort-merge, Hash join ○ Analysis of algorithm efficiency ● Query transformations ● Query optimization ○ Access paths ○ Query plan construction ○ Selectivity estimation ○ Index-only plans	Explain	KA	4
		● Database tuning: Index selection ○ Impact of indices on query performance	Develop		
DM	DBMS Internals	● DB Buffer Management ● Transaction Processing ○ Isolation Levels			

		○ ACID ○ Serializability ● Concurrency Control: [crosslist PD] ○ 2-Phase Locking ○ Deadlocks handling strategies ● Recovery Manager ○ Relation with Buffer Manager	Explain	KA	4
DM	NoSQL System	● Why NoSQL? (e.g. Impedance mismatch between Application [CRUD] and RDBMS) ● Key-Value and Document data model	Explain	KA	2
DM	Data Analytics	tbd			
DM	Data Security & Privacy	tbd			

Foundations of Programming Languages (FPL)				
Knowledge Unit	**Topic**	**Skill level**	**CS/KA Core**	**Hours**

Object-Oriented Programing	1. Imperative programming as a sunset if object-oriented programming 2. Object-oriented design a. Decomposition into objects carrying state and having behavior b. Class-hierarchy design for modeling 3. Definition of classes: fields, methods, and constructors 4. Subclasses, inheritance (including multiple inheritance), and method overriding 5. Dynamic dispatch: definition of method-call 6. Exception handling 7. Object-oriented idioms for encapsulation a. Privacy, data hiding, and visibility of class members b. Interfaces revealing only method signatures c. Abstract base classes, traits and mixins 8. Dynamic vs static properties 9. Composition vs inheritance 10. Subtyping a. Subtype polymorphism; implicit upcasts in typed languages b. Notion of behavioral replacement: subtypes acting like supertypes c. Relationship between subtyping and inheritance	Develop	CS	5
	11. Collection classes, iterators, and other common library components	Develop	KA	1

Functional Programming	1. Lambda expressions and evaluation a. Variable binding and scope rules b. Parameter passing c. Nested lambda expressions and reduction order 2. Effect-free programming a. Function calls have no side effects, facilitating compositional reasoning b. Immutable variables and data copying vs. reduction c. Use of recursion vs. loops vs. pipelining (map/reduce) 3. Processing structured data (e.g., trees) via functions with cases for each data variant a. Functions defined over compound data in terms of functions applied to the constituent pieces b. Persistent data structures 4. Using higher-order functions (taking, returning, and storing functions)	Develop	CS	4
	5. Function closures (functions using variables in the enclosing lexical environment) a. Basic meaning and definition - creating closures at run-time by capturing the environment b. Canonical idioms: call-backs, arguments to iterators, reusable code via function arguments c. Using a closure to encapsulate data in its environment d. Lazy versus eager evaluation	Explain	KA	3

Logic Programming	1. Universal vs. existential quantifiers 2. First order predicate logic vs. higher order logic 3. Expressing complex relations using logical connectives and simpler relations 4. Definitions of Horn clause, facts, goals, and subgoals 5. Unification and unification algorithm; unification vs. assertion vs expression evaluation 6. Mixing relations with functions 7. Cuts, backtracking and non-determinism 8. Closed-world vs. open-world assumptions	Explain	KA	3
Scripting	1. Error/exception handling 2. Piping 3. System commands a. Interface with operating systems 4. Environment variables 5. File abstraction and operators 6. Data structures, such as arrays and lists 7. Regular expressions 8. Programs and processes 9. Workflow	Develop	CS	2
Event-driven and Reactive Programming	1. Procedural programming vs. reactive programming: advantages of reactive programming in capturing events 2. Components of reactive programming: event-source, event signals, listeners and dispatchers, event objects, adapters, event-handlers 3. Behavior model of event-based programming 4. Canonical uses such as GUIs, mobile devices, robots, servers	Develop	CS	2

	5. Using a reactive framework a. Defining event handlers/listeners b. Parameterization of event senders and event arguments c. Externally-generated events and program-generated events 6. Separation of model, view, and controller	Develop	KA	2
Parallel and Distributed Computing	1. Safety and liveness a. Race conditions b. Dependencies/preconditions c. Fault models d. Termination 2. Programming models a. Actor models b. Procedural and reactive models c. Synchronous/asynchronous programming models d. Data parallelism 3. Semantics a. Commutativity b. Ordering c. Independence d. Consistency e. Atomicity f. Consensus 4. Execution control a. Async await b. Promises c. Threads 5. Communication and coordination a. Message-passing b. Shared memory c. cobegin-coend d. Monitors e. Channels f. Threads g. Guards	Develop	CS	3

	6. Futures 7. Language support for data parallelism such as forall, loop unrolling, map/reduce 8. Effect of memory-consistency models on language semantics and correct code generation 9. Representational State Transfer Application Programming Interfaces (REST APIs) 10. Technologies and approaches: cloud computing, high performance computing, quantum computing, ubiquitous computing 11. Overheads of message passing 12. Granularity of program for efficient exploitation of concurrency. 13. Concurrency and other programming paradigms (e.g., functional).	Explain	KA	2

Type Systems	1. A type as a set of values together with a set of operations a. Primitive types (e.g., numbers, Booleans) b. Compound types built from other types (e.g., records, unions, arrays, lists, functions, references) using set operations 2. Association of types to variables, arguments, results, and fields 3. Type safety as an aspect of program correctness 4. Type safety and errors caused by using values inconsistently given their intended types 5. Statically-typed vs dynamically-typed programming languages 6. Goals and limitations of static and dynamic typing a. Detecting and eliminating errors as early as possible 7. Generic types (parametric polymorphism) a. Definition and advantages of polymorphism: parametric, subtyping, overloading and coercion b. Comparison of monomorphic and polymorphic types c. Comparison with ad-hoc polymorphism (overloading) and subtype polymorphism d. Generic parameters and typing e. Use of generic libraries such as collections f. Comparison with ad hoc polymorphism (overloading) and subtype polymorphism g. Prescriptive vs. descriptive polymorphism h. Implementation models of polymorphic types	Develop	CS	3

	i. Subtyping			

	8. Type equivalence: structural vs name equivalence 9. Complementary benefits of static and dynamic typing a. Errors early vs. errors late/avoided b. Enforce invariants during code development and code maintenance vs. postpone typing decisions while prototyping and conveniently allow flexible coding patterns such as heterogeneous collections c. Typing rules i. Rules for function, product, and sum types d. Avoid misuse of code vs. allow more code reuse e. Detect incomplete programs vs. allow incomplete programs to run f. Relationship to static analysis g. Decidability	Develop	KA	4

Systems Programming	1. Data structures for translation, execution, translation and code mobility such as stack, heap, aliasing (sharing using pointers), indexed sequence and string 2. Direct, indirect, and indexed access to memory location 3. Run-time representation of data abstractions such as variables, arrays, vectors, records, pointer-based data elements such as linked-lists and trees, and objects 4. Abstract low-level machine with simple instruction, stack and heap to explain translation and execution 5. Run-time layout of memory: activation record (with various pointers), static data, call-stack, heap a. Translating selection and iterative constructs to control-flow diagrams b. Translating control-flow diagrams to low level abstract code c. Implementing loops, recursion, and tail calls a. Translating function/procedure calls and return from calls, including different parameter passing mechanism using an abstract machine 6. Memory management a. Low level allocation and accessing of high-level data structures such as basic data types, n-dimensional array, vector, record, and objects b. Return from procedure as automatic deallocation mechanism for local data elements in the stack c. Manual memory management: allocating, de-allocating, and reusing heap memory d. Automated memory management: garbage collection as an	Develop	CS	3

	automated technique using the notion of reachability			

Language Translation and Execution	1. Interpretation vs. compilation to native code vs. compilation to portable intermediate representation a. BNF and extended BNF representation of context-free grammar b. Parse tree using a simple sentence such as arithmetic expression or if-then-else statement c. Execution as native code or within a virtual machine 2. Language translation pipeline: syntax analysis, parsing, optional type-checking, translation/code generation and optimization, linking, loading, execution	Explain	CS	4
	3. Run-time representation of core language constructs such as objects (method tables) and first-class functions (closures) 4. Secure compiler development	Explain	KA	1

Program Abstraction and Representation	1. BNF and regular expressions 2. Programs that take (other) programs as input such as interpreters, compilers, type-checkers, documentation generators 3. Components of a language a. Definitions of alphabets, delimiters, sentences, syntax and semantics b. Syntax vs. semantics 4. Program as a set of non-ambiguous meaningful sentences 5. Basic programming abstractions: constants, variables, declarations (including nested declarations), command, expression, assignment, selection, definite and indefinite iteration, iterators, function, procedure, modules, exception handling 6. Mutable vs. immutable variables: advantages and disadvantages of reusing existing memory location vs. advantages of copying and keeping old values; storing partial computation vs. recomputation 7. Types of variables: static, local, nonlocal, global; need and issues with nonlocal and global variables 8. Scope rules: static vs. dynamic; visibility of variables; side-effects 9. Side-effects induced by nonlocal variables, global variables and aliased variables	Explain	KA	3

Total CS Core hours: 23
Total KA Core hours: 20

Graphics and Interactive Techniques (GIT)				
KU	**Topic**	**Skill**	**Core**	**Hours**

Core	● Applications ● Human vision system ● Digitization of analog data ● Standard media formats ● Color Models ● Tradeoffs between storing data and re-computing data ● Animation as a sequence of still images ● SEP related to graphics	Explain	CS	4
Basic Rendering	● Graphics pipeline. ● Affine and coordinate system transformations. ● Rendering in nature, e.g., the emission and scattering of light and its relation to numerical integration. ● Texture mapping, including minification and magnification (e.g., trilinear MIP-mapping). ● Sampling and anti-aliasing. ● Visibility and occlusion, including solutions to this problem such as depth buffering, Painter’s algorithm, and ray tracing.	Explain	KA	10
	● Forward and backward rendering (i.e., ray-casting and rasterization). ● Polygonal representation. ● Basic radiometry, similar triangles, and projection model. ● Ray tracing. ● The rendering equation. ● Simple triangle rasterization. ● Application of spatial data structures to rendering. ● Scene graphs.	Explain	KA	5
	● Generate an image with a standard API	Implement	KA	3

Visualization KA Core	● Visualization of: ○ 2D/3D scalar fields: color mapping ○ Time-varying data	Explain and Implement	KA	3
	● Visualization techniques (color mapping, dimension reduction)	Explain and Implement	KA	2
	● Perceptual and cognitive foundations that drive visual abstractions.	Explain	KA	1
	● Visualization Bias	Evaluate	KA	1
Modeling KA Core	● Surface representation/model ● Mesh representation, mesh fairing, and mesh generation techniques such as Delaunay triangulation, marching cubes ● Parametric polynomial curves and surfaces	Explain and Use	KA	2
	● Volumetric representation/model ● Volumes, voxels, and point-based representations. ● Constructive Solid Geometry (CSG) representation	Explain and Use	KA	2
	● Procedural representation/model ● Fractals ● L-Systems, cross referenced with programming languages (grammars to generated pictures). ● Generative Modeling	Explain and Use	KA	2

Shading KA Core	● Time (motion blur) and lens position (focus) and their impact on rendering ● Shadow mapping ● Occlusion culling ● Area light sources ● Hierarchical depth buffering ● Non-photorealistic rendering	Explain and Use	KA	6
Computer Animation KA Core	● Principles of Animation (Squash and Stretch, Timing, Anticipation, Staging, Follow Through and Overlapping Action, Straight Ahead Action and Pose-to-Pose Action, Slow In and Out, Arcs, Exaggeration, and Appeal) ● Key-frame animation	Explain and Use	KA	2
	● Forward and inverse kinematics	Explain and Use	KA	2
	● Transforms: ○ ● Translations ○ ● Scale / Shear ○ ● Rotations ● Camera animation ○ ● Look at ○ ● Focus	Explain and Implement	KA	2
Simulation KA Core	● Particle systems	Explain and implement	KA	2
	● Collision detection and response	Explain and Implement	KA	2
	● Grid based fluids	Explain and Implement	KA	2
Immersion KA Core	● Define and distinguish VR, AR, and MR ● Applications in medicine, simulation, and training, and visualization	Explain	KA	1

	● Stereoscopic display ● Viewer tracking ○ Inside out vs Outside In ○ Head / Body / Hand / tracking ● Visibility computation	Explain and Implement	KA	3
	● Safety in immersive applications ● Accessibility in immersive applications. ● Ethics/privacy in immersive applications.	Explain and Evaluate	KA	2
Interaction KA Core	● Event Driven Programming (Shared with SPD Interactive, SPD Game Development) ● Graphical User Interface	Explain and Implement	KA	4
	● Accessibility in GUI Design (shared with HCI)	Explain and Evaluate	KA	2
Image Processing	● Convolution filters	Explain and implement	KA	2
	● Convolutional Neural Networks.(shared with AI:Machine Learning)	Explain and Implement	KA	2
	● Histograms ● Fourier and/or Cosine Transforms	Explain and Implement	KA	2
Physical Computing KA Core	● Communication with the physical world (shared with SPD/Embedded Systems, PL/Embedded Systems) ○ Acquisition of data from sensors ○ Driving external actuators	Explain and Implement	KA	1
	● Event driven programming (shared with GIT: Interaction)	Explain and Implement	KA	1

	● Connection to physical artifacts ○ Computer Aided Design ○ Computer Aided Manufacturing ○ Fabrication ■ prototyping (shared with HCI) ■ Additive (3D printing) ■ Subtractive (CNC milling) ■ Forming (vacuum forming)	Explain, evaluate, and Implement an example	KA	3
	● Internet of Things ○ Network connectivity ○ Wireless communication	Explain	KA	1

Missing for HCI

Networking and Communication (NC)					
KA	**KU**	**Topic**	**Skill**	**Core**	**Hours**
NC	Introduction	● Importance of networking in contemporary computing, and associated challenges.	Explain	CS	3
		● Organization of the Internet ○ Users, ○ Internet Service Providers ○ Autonomous systems ○ Content providers ○ Content delivery networks	Explain		
		● Switching techniques ○ Circuit Switching ○ Packet Switching	Evaluate		
		● Layers and their roles. ○ Application ○ Transport ○ Network ○ Datalink ○ Physical	Explain		
		● Layering principles ○ Encapsulation	Explain		

		○ Hourglass model			
		● Network elements ○ Routers ○ Switches ○ Hubs ○ Access points ○ Hosts	Explain		
		● Basic queueing concepts ○ Relationship with latency ○ Relationship with Congestion ○ Relationship with Service levels	Explain		
NC	Networked Applications	● Naming and address schemes. ○ DNS ○ IP addresses ○ Uniform Resource Identifiers	Explain	CS	4
		● Distributed application paradigms ○ Client/server ○ Peer-to-peer ○ Cloud ○ Edge ○ Fog	Evaluate		
		● Diversity of networked application demands ○ Latency ○ Bandwidth ○ Loss tolerance	Explain		
		● Application-layer development using one or more protocols: ○ HTTP ○ SMTP ○ POP3	Develop		
		● Interactions with TCP, UDP, and Socket APIs.	Explain		
NC	Reliability Support	● Unreliable delivery ○ UDP ○ Other	Explain	KA	6
		● Principles of reliability ○ Delivery without loss ○ Duplication ○ Out of order	Develop		

		● Error control ○ Retransmission ○ Error correction	Evaluate		
		● Flow control ○ Stop and wait ○ Window based	Develop		
		● Congestion control ○ Implicit congestion notification ○ Explicit congestion notification	Explain		
		● TCP and performance issues ○ Tahoe ○ Reno ○ Vegas ○ Cubic ○ QUIC	Evaluate		
NC	Routing and Forwarding	● Routing paradigms and hierarchy ○ Intra/inter domain ○ Centralized and decentralized ○ Source routing ○ Virtual circuits ○ QoS	Evaluate	KA	4
		● Forwarding methods ○ Forwarding tables ○ Matching algorithms	Apply		
		● IP and Scalability issues ○ NAT ○ CIDR ○ BGP ○ Different versions of IP	Explain		
NC	Single Hop Communication	● Introduction to modulation, bandwidth, and communication media.	Explain	KA	3
		● Encoding and Framing.	Evaluate		
		● Medium Access Control (MAC) ○ Random access ○ Scheduled access	Evaluate		
		● Ethernet	Explain		
		● Switching	Apply		

		● Local Area Network Topologies (e.g. data center networks)	Explain		
NC	Network Security	● General intro about security [Shared with Security] ○ Threats ○ Vulnerabilities ○ Countermeasures	Explain	KA	4
		● Network specific threats and attack types [Shared with Security] ○ Denial of service ○ Spoofing ○ Sniffing ○ Traffic redirection ○ Man-in-the-middle ○ Message integrity attacks ○ Routing attacks ○ Traffic analysis	Explain		
		● Countermeasures [Shared with Security] ○ Cryptography (e.g. SSL, symmetric/asymmetric). ○ Architectures for secure networks (e.g., secure channels, secure routing protocols, secure DNS, VPNs, DMZ, Zero Trust Network Access, hyper network security, anonymous communication protocols, isolation) ○ Network monitoring, intrusion detection, firewalls, spoofing and DoS protection, honeypots, tracebacks, BGP Sec.	Explain		
NC	Mobility	● Principles of cellular communication (e.g. 4G, 5G)	Explain	KA	3
		● Principles of Wireless LANs (mainly 802.11)	Explain		
		● Device to device communication [Shared with SPD]	Explain		
		● Multihop wireless networks	Explain		
		● Examples (e.g ad hoc networks, opportunistic, delay tolerant)	Explain		

NC	Emerging Topics	• Middleboxes (e.g. filtering, deep packet inspection, load balancing, NAT, CDN)	Explain	KA	4
		• Virtualization (e.g. SDN, Data Center Networks)	Explain		
		• Quantum Networking (e.g. Intro to the domain, teleportation, security, Quantum Internet)	Explain		

Operating Systems (OS)

	KU	Topics	Skill	Core	Hours
OS	Role and Purpose of Operating Systems	• Operating system as mediator between general purpose hardware and application-specific software [Overlap with PL] • Universal operating system functions • Extended and/or specialized operating system functions • Design issues (e.g. efficiency, robustness, flexibility, portability, security, compatibility, power, safety) • Influences of security, networking, multimedia, parallel and distributed computing • Overarching concern of security/protection: Neglecting to consider security at every layer creates an opportunity to inappropriately access resources.	Explain	CS	3
OS	Principles of Operating System	• Operating system software design and approaches • Abstractions, processes, files and resources • Concept of system calls and links to application program interfaces (APIs) [Shared with AR] • The evolution of the link between hardware architecture and the	Explain	CS	3

		operating system functions [Shared with AR] ● Protection of resources means protecting some machine instructions/functions[Shared with AR] ● Leveraging interrupts from hardware level: service routines and implementations[Shared with AR] ● Concept of user/system state and protection, transition to kernel mode using system calls [Shared with AR] ● Mechanism for invoking of system calls, the corresponding mode and context switch and return from interrupt [Shared with AR]			
OS	Concurrency (non-core topics not listed)	● Thread abstraction relative to concurrency ● Race conditions, critical sections (role of interrupts if needed) ● Deadlocks and starvation ● Multiprocessor issues (spin-locks, reentrancy)	Explain	CS	3
		● Thread creation, states, structures ● Thread APIs ● Deadlocks and starvation (necessary conditions/mitigations) ● Implementing thread safe code (semaphores, mutex locks, cond vars) ● Race conditions in shared memory [Shared with PD]	Apply	KA	3
OS	Scheduling (non-core topics not listed)	● Preemptive and non-preemptive scheduling ● Timers (e.g. building many timers out of finite hardware timers). ● Schedulers and policies ● Concepts of SMP/multiprocessor scheduling [Shared with AR]	Explain	KA	3
OS	Process Model	● Processes and threads relative to virtualization-Protected memory, process state, memory isolation, etc	Explain	KA	3

		● Memory footprint/segmentation (stack, heap, etc) ● Creating and loading executables and shared libraries ● Dispatching and context switching ● Interprocess communication			
OS	Management (non-core topics not listed)	● Review of physical memory, address translation and memory management hardware ● Impact of memory hierarchy including cache concept, cache lookup, etc on operating system mechanisms and policy ● Logical and physical addressing ● Concepts of paging, page replacement, thrashing and allocation of pages and frames ● Allocation/deallocation/storage techniques (algorithms and data structure) performance and flexibility ● Memory Caching and cache coherence ● Security mechanisms and concepts in memory management including sandboxing, protection, isolation, and relevant vectors of attack	Explain	KA	4
OS	Protection and Safety (Overlap with Security)	● Overview of operating system security mechanisms ● Attacks and antagonism (scheduling, etc.) ● Review of major vulnerabilities in real operating systems ● Operating systems mitigation strategies such as backups	Apply	CS	3
		● Policy/mechanism separation ● Security methods and devices ● Protection, access control, and authentication	Apply	KA	1
OS	Device Management	● Buffering strategies ● Direct Memory Access and Polled I/O, Memory-mapped I/O Historical and contextual - Persistent storage	Explain	KA	2

	(non-core not listed) [Shared memory in AR]	device management (magnetic, SSD, etc.)			
OS	File Systems API and Implementation (Historical significance but may play decreasing role moving forward)	● Concept of a file ● File system mounting ● File access control ● File sharing ● Basic file allocation methods ● File system structures comprising file allocation including various directory structures and methods for uniquely identifying files (name, identified or metadata storage location) ● Allocation/deallocation/storage techniques (algorithms and data structure) impact on performance and flexibility ● Free space management ● Implementation of directories to segment and track file location	Explain	KA	2
OS	Advanced File Systems (non-core topics not listed)	● File systems: partitioning, mount/unmount, virtual file systems ● In-depth implementation techniques ● Memory-mapped files ● Special-purpose file systems ● Naming, searching, access, backups ● Journaling and log-structured file systems	Explain	KA	2
OS	Virtualization(non-core topics not listed)	● Using virtualization and isolation to achieve protection and predictable performance ● Advanced paging and virtual memory ● Virtual file systems and virtual devices ● Thrashing ● Containers	Explain	KA	2
OS	Real-time and	● Process and task scheduling ● Deadlines and real-time issues	Explain	KA	1

	Embedded Systems (non-core topics not listed)	● Low-latency/soft real-time" vs "hard real time" [shared with PL]			
OS	Fault Tolerance (non-core topics not listed)	● Reliable and available systems ● Software and hardware approaches to address tolerance (RAID)	Explain	KA	1
OS	Social, Ethical and Professional topics	● Open source in operating systems ● End-of-life issues with sunsetting operating systems [Covered in SE]	Explain	KA	4

Parallel and Distributed Computing (PDC)					
KA	KU	Topic	Skill	Core	Hours
PDC	Programs and Executions	Definitions and Properties ● Parallelizable actions ● Ordering among actions; happens-before relations ● Independence: determining when ordering doesn’t matter in terms of commutativity, dependencies, preconditions ● Consistency: Agreement about values and predicates; races, atomicity, consensus ● Ensuring ordering when necessary in parallel programs ● Places: Physical devices executing parallel actions (parties) ● Faults arising from failures in parties or communication	Explain	CS	1
PDC	Programs and Executions	Starting parallel actions ● Placement: arranging that the action be performed (eventually) by a designated party	Explain	CS	1

		● Procedural: Enabling multiple actions to start at a given program point ● Reactive: Enabling upon an event ● Dependent: Enabling upon completion of others			
PDC	Programs and Executions	Underlying mappings and mechanisms. **One or more of:** 1. CPU data- and instruction-level- parallelism 2. SIMD and heterogeneous data parallelism 3. Multicore scheduled concurrency, tasks, actors 4. Clusters, clouds; elastic provisioning 5. Distributed systems with unbounded participants 6. Emerging technologies such as quantum computing and molecular computing	Explain	KA	2
PDC	Communication	Fundamentals ● Media ○ Varieties: channels (message passing or IO), shared memory, heterogeneous, data stores ○ Reliance on the availability and nature of underlying hardware, connectivity, and protocols; language support, emulation ● Channels ○ Explicit party-to-party communication; naming ○ APIs: sockets, architectural and language-based channel constructs ● Memory ○ Architectures in which parties directly communicate only with memory at given addresses ○ Consistency: Bitwise atomicity limits, coherence, local ordering ○ Memory hierarchies, locality: caches, latency, false-sharing ○ Heterogeneous Memory using multiple memory stores, with explicit data transfer across them; for example, GPU local and shared memory, DMA	Explain	CS	2

		○ Multiple layers of sharing domains, scopes and caches ● Data Stores ○ Cooperatively maintained structured data implementing maps, sets, and related ADTs ○ Varieties: Owned, shared, sharded, replicated, immutable, versioned			
PDC	Communication	Programming with Communication ● Using channel, socket, and/or remote procedure APIs ● Using shared memory constructs in a given language	Develop	CS	1
PDC	Communication	Properties and Extensions. **One or more of:** ● Media ○ Topologies: Unicast, Multicast, Mailboxes, Switches; Routing ○ Concurrency properties: Ordering, consistency, idempotency, overlapping with computation ○ Reliability: transmission errors and drops. ○ Data formats, marshaling ○ Protocol design: progress guarantees, deadlocks ○ Security: integrity, privacy, authentication, authorization. ○ Performance Characteristics: Latency, Bandwidth (throughput), Contention (congestion), Responsiveness (liveness). ○ Applications of Queuing Theory to model and predict performance ● Channels ○ Policies: Endpoints, Sessions, Buffering, Saturation response (waiting vs dropping), Rate control ○ Program control for sending (usually procedural) vs	Explain	KA	6

		receiving.(usually reactive or RPC-based) ○ Formats, marshaling, validation, encryption, compressIon ○ Multiplexing and demultiplexing in contexts with many relatively slow IO devices or parties; completion-based and scheduler-based techniques; async-await, select and polling APIs. ○ Formalization and analysis; for example using CSP ● Memory ○ Memory models: sequential and release/acquire consistency ○ Memory management; including reclamation of shared data; reference counts and alternatives ○ Bulk data placement and transfer; reducing message traffic and improving locality; overlapping data transfer and computation; impact of data layout such as array-of-structs vs struct-of-arrays ○ Emulating shared memory: distributed shared memory, RDMA ● Data Stores ○ Consistency: atomicity, linearizability, transactionality, coherence, causal ordering, conflict resolution, eventual consistency, blockchains, ○ Faults and partial failures; voting; protocols such as Paxos and Raft ○ Security and trust: Byzantine failures, proof of work and alternatives			
PDC	Coordination	Fundamentals ● Dependent actions	Explain	CS	2

		○ Execution control when one activity’s initiation or progress depends on actions of others ○ Completion-based: Barriers, joins ○ Data-enabled: Produce-Consumer designs ○ Condition-based: Polling, retrying, backoffs, helping, suspension, queueing, signaling, timeouts ○ Reactive: enabling and triggering continuations ● Progress ○ Dependency cycles and deadlock; monotonicity of conditions ● Atomicity ○ Atomic instructions, enforced local access orderings ○ Locks and mutual exclusion ○ Deadlock avoidance: ordering, coarsening, randomized retries; encapsulation via lock managers ○ Common errors: failing to lock or unlock when necessary, holding locks while invoking unknown operations, deadlock			
PDC	Coordination	Programming with coordination ● Controlling termination ● Using locks, barriers, and other synchronizers in a given language; maintaining liveness without introducing races ● Using transactional APIs in a given framework	Develop	CS	1
PDC	Coordination	Properties and extensions. **One or more of:** ● Progress ○ Properties including lock-free, wait-free, fairness, priority scheduling; interactions with consistency, reliability	Explain	KA	6

		○ Performance: contention, granularity, convoying, scaling ○ Non-blocking data structures and algorithms ● Atomicity ○ Ownership and resource control ○ Lock variants: sequence locks, read-write locks; reentrancy; tickets ○ Transaction-based control: Optimistic and conservative ○ Distributed locking: reliability ● Interaction with other forms of program control ○ Alternatives to barriers: Clocks; Counters, virtual clocks; Dataflow and continuations; Futures and RPC; Consensus-based, Gathering results with reducers and collectors ○ Speculation, selection, cancellation; observability and security consequences ○ Resource-based: Semaphores and condition variables ○ Control flow: Scheduling computations, Series-parallel loops with (possibly elected) leaders, Pipelines and Streams, nested parallelism. ○ Exceptions and failures. Handlers, detection, timeouts, fault tolerance, voting			
PDC	Software Engineering	Safety, liveness and performance requirements 7. Temporal logic constructs to express “always” and “eventually” 8. Metrics for throughput, responsiveness, latency, availability, energy consumption, scalability, resource usage, communication costs, waiting and rate control, fairness; service level agreements	Explain	CS	1
PDC	Software Engineering	Identifying, testing for, and repairing violations 9. Common forms of errors: failure to ensure necessary ordering (race errors), atomicity	Evaluate	CS	1

		(including check-then-act errors), or termination (livelock).			
PDC	Software Engineering	Specification and Evaluation. **One or more of:** 10. Formal Specification a. Extensions of sequential requirements such as linearizability; protocol, session, and transactional specifications b. Use of tools such as UML, TLA, program logics c. Security: safety and liveness in the presence of hostile or buggy behaviors by other parties; required properties of communication mechanisms (for example lack of cross-layer leakage), input screening, rate limiting 11. Static Analysis a. For correctness, throughput, latency, resources, energy b. dag model analysis of algorithmic efficiency (work, span, critical paths) 12. Empirical Evaluation a. Testing and debugging; tools such as race detectors, fuzzers, lock dependency checkers, unit/stress/torture tests, continuous integration, continuous deployment, and test generators, b. Measuring and comparing throughput, overhead, waiting, contention, communication, data movement, locality, resource usage, behavior in the presence of too many events, clients, threads. 13. Application domain specific analyses and evaluation techniques	Evaluate	KA	3
PDC	Algorithms and applications	Expressing and implementing parallel and distributed algorithms 14. Implementing concepts in given languages and frameworks to initiate activities (for example threads), use shared memory	Develop	CS	1

		constructs, and channel, socket, and/or remote procedure call APIs			
PDC	Algorithms and applications	Survey of primary categories and algorithmic domains (listed below).	Explain	CS	1
PDC	Algorithms and applications	Algorithmic Domains. **One or more of:** 15. Linear Algebra: Vector and Matrix operations, numerical precision/stability, applications in data analytics and machine learning 16. Data processing: sorting, searching and retrieval, concurrent data structures 17. Graphs, search, and combinatorics: Marking, edge-parallelization, bounding, speculation, network-based analytics 18. Modeling and simulation: differential equations; randomization, N-body problems, genetic algorithms 19. Logic: SAT, concurrent logic programming 20. Graphics and computational geometry: Transforms, rendering, ray-tracing 21. Resource Management: Allocating, placing, recycling and scheduling processors, memory, channels, and hosts. Exclusive vs shared resources. Static, dynamic and elastic algorithms, Batching, prioritization, partitioning, decentralization via work-stealing and related techniques 22. Services: Implementing Web APIs, Electronic currency, transaction systems, multiplayer games.	Develop	KA	9

Software Development Fundamentals (SDF)					
KA	**KU**	**Topic**	**Skill**	**Core**	**Hours**
SDF	Fundamental Programming Concept	● Basic concepts such as variables, primitive data types, and expression evaluation. ● How imperative programs work: state and state transitions on execution of statements, flow of control	Develop	CS	18

	s	● Basic constructs such as assignment statements, conditional and iterative structures and flow of control ● Key modularity constructs such as functions/methods and classes, and related concepts like parameter passing, scope, abstraction, data encapsulation, etc. ● Input and output using files, console, and APIs ● Structured data types available in the chosen programming language like sequences ● Libraries and frameworks provided by the language (when/where applicable) ● Recursion			
SDF	Fundamental Programming Concepts	● Basic concept of programming errors, testing, and debugging. Dealing with compile time and runtime errors ● Reading and understanding code	Evaluate, Apply	CS	2
SDF	Fundamental Data Structures	● Standard abstract data types such as lists, stacks, queues, sets, and maps/dictionaries [Shared with: AL] ● Strings and string processing	Develop	CS	10
SDF	Fundamental Data Structures	● When and how to use standard data structures ● Performance implications of choice of data structure(s)	Evaluate	CS	2
SDF	Algorithms	● Concept of algorithm and notion of algorithm efficiency ● Common algorithms like: Sorting, Searching, Tree traversal, Graph traversal, etc. [Shared with AL]	Explain	CS	4
SDF	Algorithms	● Assessing the time/space efficiency of algorithms through measurement [Shared with AL]	Evaluate	CS	2

SDF	Software Development practices	● Programming style that improves readability [Shared with SE]	Evaluate	CS	1
SDF	Software Development practices	● Basic unit testing (using suitable frameworks) including test case design [Shared with SE]	Develop	CS	2
SDF	Software Development practices	● Use of a general-purpose IDE, including its debugger (which can be also used to strengthen some programming concepts)	Apply	CS	2

Software Engineering (SE)					
KA	**KU**	**Topic**	**Skill**	**Core**	**Hours**
SE	Teamwork	● Effective communication ● Common causes of team conflict, and approaches for conflict resolution ● Cooperative programming ● Roles and responsibilities in a software team ● Team processes ● Importance of team diversity	Evaluate	CS	2
SE	Teamwork	● Interfacing with stakeholders, as a team ● Risks associated with physical, distributed, hybrid and virtual teams	Explain	KA	2
SE	Tools and Environments	● Software configuration management and version control	Evaluate	CS	1
SE	Tools and	● Release management ● Testing tools including static and dynamic			

	Environments	analysis tools ● Software process automation ● Design and communication tools (docs, diagrams, common forms of design diagrams like UML) ● Tool integration concepts and mechanisms ● Use of modern IDE facilities - debugging, refactoring, searching/indexing, etc.	Explain	KA	3
SE	Product Requirements	● Describe functional requirements using, for example, use cases or user stories ● Properties of requirements including consistency, validity, completeness, and feasibility ● Requirements elicitation ● Non-functional requirements, for example, security, usability, or performance (aka Quality Attributes) ● Risk identification and management ● Communicating and/or formalizing requirement specifications	Apply	KA	2
SE	Software Design	● System design principles ● Software architecture ● Programming in the large vs. programming in the small ● Code smells and other indications of code quality, distinct from correctness.	Explain	CS	1
SE	Software Design	● API design principles ● Identifying and codifying data invariants and time invariants ● Structural and behavioral models of software designs ● Data design ● Requirement traceability	Apply	KA	4
SE	Software Construction	● Practical small-scale testing ● Documentation	Apply	CS	1
SE	Software Design	● Coding Style ● “Best Practices” for coding ● Debugging ● Use of libraries and frameworks developed by others	Apply	KA	3

SE	Software Verification and Validation	● Verification and validation concepts ● Why testing matters ● Testing objectives ● Test kinds ● Stylistic differences between tests and production code	Explain	CS	1
SE SDF	Software Verification and Validation	● Test planning and generation ● Test development (see SDF) ● Verification and validation in the development cycle ● Domain specific verification and validation challenges	Explain	KA	4
SE	Refactoring and Code Evolution	● Hyrum's Law ● Backward Compatibility ● Refactoring ● Versioning	Explain	KA	1
SE SEP SF	Software Reliability	● Concept of reliability ● Identifying reliability requirements (see SEP) ● Software failures vs. defect injection/detection ● Software reliability, system reliability and failure behavior (cross-reference SF/Reliability Through Redundancy) ● Defect injection and removal cycle , and different approaches for defect removal ● Compare the "error budget" approach to reliability with the "error-free" approach, and identify domains where each is relevant	Explain	KA	4

Missing for SEC

Society, Ethics and Professionalism (SEP)					
KA	**KU**	**Topic**	**Skill**	**Core**	**Hours**
SEP	Social Context	1. Social implications of computing in a hyper-networked world where the capabilities of artificial intelligence are rapidly evolving	Evaluate	CS	3

		2. Impact of social media and artificial intelligence on individual well-being, political ideology, and cultural ideology 3. Impact of involving computing technologies, particularly artificial intelligence, biometric technologies and algorithmic decision-making systems, in civic life (e.g., facial recognition technology, biometric tags, resource distribution algorithms, policing software)			
SEP	Social Context	1. Growth and control of the internet, computing, and artificial intelligence 2. Often referred to as the digital divide, differences in access to digital technology resources and its resulting ramifications for gender, class, ethnicity, geography, and/or developing nations 3. Accessibility issues, including legal requirements and dark patterns 4. Context-aware computing	Explain	KA	2
SEP	Methods for Ethical Analysis	1. Avoiding fallacies and misrepresentation in argumentation 2. Ethical theories and decision-making (philosophical and social frameworks and epistemologies) 3. Recognition of the role culture plays in our understanding, adoption, design, and use of computing technology	Apply	CS	2
SEP	Methods for Ethical Analysis	1. Professional checklists 2. Evaluation rubrics 3. Stakeholder analysis 4. Standpoint theory	Develop	KA	1
SEP	Professional Ethics	1. Community values and the laws by which we live 2. The nature of professionalism including care, attention and discipline, fiduciary responsibility, and mentoring 3. Keeping up-to-date as a computing professional in terms of familiarity, tools, skills, legal and professional frameworks as well as the ability to self-assess and progress in the computing field	Evaluate	CS	2

		4. Professional certification, codes of ethics, conduct, and practice, such as the ACM/IEEE-CS, SE, AITP, IFIP and international societies 5. Accountability, responsibility and liability (e.g., software correctness, reliability and safety, as well as ethical confidentiality of cybersecurity professionals) 6. Introduction to theories describing the human creation and use of technology including instrumentalism, sociology of technological systems, disability justice, neutrality thesis, pragmatism, utilitarianism, and decolonial theories 7. Develop strategies for recognizing and reporting designs, systems, software, and professional conduct (or their outcomes) that may violate law or professional codes of ethics			
SEP	Professional Ethics	1. The role of the computing professional in public policy 2. Maintaining awareness of consequences 3. Ethical dissent and whistle-blowing 4. The relationship between regional culture and ethical dilemmas 5. Dealing with harassment and discrimination 6. Forms of professional credentialing 7. Acceptable use policies for computing in the workplace 8. Ergonomics and healthy computing environments 9. Time to market and cost considerations versus quality professional standards	Explain	KA	2
SEP	Intellectual Property	1. Philosophical foundations of intellectual property 2. Intellectual property rights 3. Intangible digital intellectual property (IDIP) 4. Legal foundations for intellectual property protection	Explain	CS	1
SEP	Intellect	1. Digital rights management	Apply	KA	1

	ual Property	2. Copyrights, patents, trade secrets, trademarks 3. Plagiarism 4. Foundations of the open source movement 5. Software piracy			
SEP	Privacy and Civil Liberties	1. Privacy implications of widespread data collection for transactional databases, data warehouses, surveillance systems, and cloud computing 2. Ramifications of differential privacy 3. Technology-based solutions for privacy protection 4. Civil liberties and cultural differences	Explain	CS	2
SEP	Privacy and Civil Liberties	1. Philosophical foundations of privacy rights 2. Legal foundations of privacy protection in relevant jurisdictions 3. Privacy legislation in areas of practice 4. Freedom of expression and its limitations	Assess	KA	1
SEP	Commu nication	1. Interpreting, summarising, and synthesising technical material, including source code and documentation 2. Writing effective technical documentation and materials (tutorials, reference materials, API documentation) 3. Identifying, describing, and employing (clear, polite, concise) oral, written, and electronic team and group communication. 4. Understanding and enacting awareness of audience in communication by communicating effectively with different customers, stakeholders, and leadership 5. Utilising collaboration tools 6. Recognizing and avoiding the use of rhetorical fallacies when resolving technical disputes 7. Understanding accessibility and inclusivity requirements for addressing professional audiences	Apply	CS	2
SEP	Commu nication	1. Demonstrate cultural competence in written and verbal communication	Apply	KA	1

		2. Using synthesis to concisely and accurately convey tradeoffs in competing values driving software projects including technology, structure/process, quality, people, market and financial 3. Use writing to solve problems or make recommendations in the workplace, such as raising ethical concerns or addressing accessibility issues			
SEP	Sustainability	1. Being a sustainable practitioner by taking into consideration environmental, social, and cultural impacts of implementation decisions (e.g., algorithmic bias/outcomes, , economic viability, and resource consumption) 2. Explore local/regional/global social and environmental impacts of computing systems use and disposal (e-waste) 3. Discuss the tradeoffs involved in proof-of-work and proof-of-stake algorithm	Apply	CS	1
SEP	Sustainability	1. Guidelines for sustainable design standards 2. Systemic effects of complex computer-mediated phenomena (e.g., social media, offshoring, remote work) 3. 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 4. Conduct research on applications of computing to environmental issues, such as energy, pollution, resource usage, recycling and reuse, food management / production, and others 5. How the sustainability of software systems are interdependent with social systems, including the knowledge and skills of its users, organizational processes and policies, and its societal context (e.g., market forces, government policies)	Evaluate	KA	1

SEP	History	1. Age I: Prehistory—the world before ENIAC (1946): Ancient analog computing (Stonehenge, Antikythera mechanism, Salisbury Cathedral clock, etc.), Euclid, Lovelace, Babbage, Gödel, Church, Turing, pre-electronic (electro-mechanical and mechanical) hardware 2. Age II: Early modern (digital) computing - ENIAC, UNIVAC, Bombes (Bletchley Park codebreakers), mainframes, etc. 3. Age III: Modern (digital) computing - PCs, modern computer hardware, Moore's Law 4. Age IV: Internet - networking, internet architecture, browsers and their evolution, standards, big players (Google, Amazon, Microsoft, etc.), distributed computing 5. Age V: Cloud - smart phones (Apple, Android, and minor ones), cloud computing, remote servers, software as a service (SaaS), security and privacy, social media 6. Age VI: Emerging AI-assisted technologies including decision making systems, recommendation systems, generative AI and other machine learning driven tools and technologies	Explain	KA	1
SEP	Economies of Computing	1. Economies of providers: regulated and unregulated, monopolies and open-market. "Walled Gardens" in tech environments 2. The knowledge and attention economies 3. Effect of skilled labor supply and demand on the quality of computing products 4. Pricing strategies in the computing domain: subscriptions, planned obsolescence, software licenses, open-source, free software 5. Outsourcing and off-shoring software development; impacts on employment and on economics 6. Consequences of globalization for the computer science profession and users 7. Differences in access to computing resources and the possible effects thereof 8. Automation and its effect on job markets, developers, and users	Explain	KA	1

		9. Economies of scale, startups, entrepreneurship, philanthropy 10. How computing is changing personal finance: Blockchain and cryptocurrencies, mobile banking and payments, SMS payment in developing regions, etc.			
SEP	Security Policies, Laws and Computer Crimes	1. Examples of computer crimes and legal redress for computer criminals 2. Social engineering, computing-enabled fraud, and recovery 3. Identify what constitutes computer crime, such as Issues surrounding the misuse of access and breaches in security 4. Motivations and ramifications of cyber terrorism and criminal hacking, “cracking” 5. Effects of malware, such as viruses, worms and Trojans	Explain	CS	2
SEP	Security Policies, Laws and Computer Crimes	1. Crime prevention strategies 2. Security policies	Apply	KA	1
SEP	Inclusion, Diversity, Equity, and Accessibility	1. How identity impacts and is impacted by computing environments (academic and professional) and technologies 2. The benefits of diverse development teams and the impacts of teams that are not diverse. 3. Inclusive language and charged terminology, and why their use matters 4. Inclusive behaviors and why they matter 5. Technology and accessibility 6. How computing professionals, via the software they create, can influence and impact justice, diversity, equity, and inclusion both positively and negatively	Explain	CS	2
SEP	Inclusion, Diversity,	1. Highlight experts (practitioners, graduates, and upper level students) who reflect the identities of the classroom and the world	Evaluate	KA	2

	Equity, and Accessibility	2. Benefits of diversity and harms caused by a lack of diversity 3. Historic marginalization due to technological supremacy and global infrastructure challenges to equity and accessibility			

Systems Fundamentals (SF)					
KA	**KU**	**Topic**	**Skill**	**Core**	**Hours**
SF	Overview of Computer Systems	● 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 expressions, minimization, sum of product forms ● Programming abstractions, interfaces, use of libraries ● Distinction between application and OS services, remote procedure call ● Application-OS interaction ● Basic concept of pipelining, overlapped processing stages ● Basic concept of scaling: going faster vs. handling larger problems	Explain	CS	3
SF	Basic Concepts	● Digital vs. Analog/Discrete vs. Continuous Systems ● Simple logic gates, logical expressions, Boolean logic simplification ● Clocks, State, Sequencing ● State and state transition (e.g., starting state, final state, life cycle of states) ● Finite state machines (e.g., NFA, DFA) ● Combinational Logic, Sequential Logic, Registers, Memories ● Computers and Network Protocols as examples of State Machines	Apply	CS	4

SF	Resource Allocation and Scheduling	● Different types of resources (e.g., processor share, memory, disk, net bandwidth) ● Common scheduling algorithms (e.g., first-come-first-serve scheduling, priority-based scheduling, fair scheduling and preemptive scheduling) ● Advantages and disadvantages of common scheduling algorithms	Explain	CS/ KA	1/2
SF	System Performance	● Latencies in computer systems o Speed of light and computers (one foot per nanosecond vs. one GHz clocks) o Memory vs. disk latencies vs. across the network memory ● Caches and the effects of spatial and temporal locality on performance in processors and systems ● Caches and cache coherency in databases, operating systems, distributed systems, and computer architecture ● Introduction into the processor memory hierarchy and the formula for average memory access time ● Rationale of virtualization and isolation: protection and predictable performance ● Levels of indirection, illustrated by virtual memory for managing physical memory resources ● Methods for implementing virtual memory and virtual machines	Apply	CS/ KA	2/2
SF	Performance Evaluation	● Performance figures of merit ● Workloads and representative benchmarks, and methods of collecting and analyzing performance figures of merit ● CPI (Cycles per Instruction) equation as tool for understanding tradeoffs in the design of instruction sets, processor pipelines, and memory system organizations.	Evaluate	CS/ KA	2/2

		● Amdahl's Law: the part of the computation that cannot be sped up limits the effect of the parts that can ● Analytical tools to guide quantitative evaluation ● Order of magnitude analysis (Big O notation) ● Analysis of slow and fast paths of a system ● Events on their effect on performance (e.g., instruction stalls, cache misses, page faults) ● Understanding layered systems, workloads, and platforms, their implications for performance, and the challenges they represent for evaluation ● Microbenchmarking pitfalls			
SF	System Reliability	● Distinction between bugs and faults ● Reliability through redundancy: check and retry ● Reliability through redundancy: redundant encoding (error correction codes, CRC, FEC) ● Reliability through redundancy: duplication/mirroring/replicas ● Other approaches to reliability	Evaluate	CS/KA	2/1
SF	System Security	● Common system security issues (e.g., virus, denial-of-service attack and eavesdropping) ● Countermeasures ○ Cryptography ○ Security architecture ○ Intrusion detection systems, firewalls	Evaluate	CS/KA	2/1
SF	**System Design**	● Common criteria of system design (e.g., liveness, safety, robustness, scalability and security) ● Designs of representative systems (e.g., Apache web server, Spark and Linux)	Design	CS/KA	2/1

Specialized Platform Development (SPD)

KA	KU	Topic	Skill	Core	Hours
	Common Aspects	● Overview of platforms (e.g., Web, Mobile, Game, Robot, Embedded, and Interactive) ● Programming via platform-specific Application Programming Interface (API) vs traditional application construction ● Overview of Platform Languages (e.g., Kotlin, Swift, C#, C++, Java, JavaScript, HTML5) ● Programming under platform constraints (e.g., available development tools, development) ● Techniques for learning and mastering a platform-specific programming language.	Apply	CS	2
	Web Platforms	● Web programming languages (e.g., HTML5, JavaScript, PHP, CSS) ● Web platforms, frameworks, or meta-frameworks ● Software as a Service (SaaS) ● Web standards such as document object model, accessibility ● Security and Privacy considerations ● Analyzing requirements for web applications ● Computing services (e.g., Amazon AWS, Microsoft Azure) ● Data management ● Architecture ● Storage Solutions	Apply	CS	3
	Mobile Platforms	● Development ● Mobile platform constraints ● Access ● Mobile computing affordances ● Specification and Testing ● Asynchronous computing	Apply	CS	3
	Robot Platforms	● Types of robotic platforms and devices ● Sensors, embedded computation, and effectors (actuators) ● Robot-specific languages and libraries ● Robotic platform constraints and design considerations ● Interconnections with physical or simulated systems ● Robotics	Apply	KA	3
	Embedded Platforms	● Introduction to the Unique Characteristics of Embedded Systems. ● Safety considerations and safety analysis ● Sensors and Actuators ● Embedded programming ● Real-time resource management ● Analysis and Verification ● Application Design	Apply	KA	3

	Game Platforms	● Historic and Contemporary Platforms for Games ● Social, Legal, and Ethical Considerations for Game Platforms ● Real-time Simulation and Rendering Systems ● Game Development Tools and Techniques ● Game Design	Apply	KA	4
	Interactive	● Data Analysis Platforms ● Data Visualizations ● Creative coding ● Quantum Computing Platforms ● Language Models (LLMs) ● Supporting math studies ● Supporting humanities studies	Apply	KA	2
	SEP	● TBD		KA	3

Course Packaging by Competency Area

Software

Courses that span Software Development Fundamentals (SDF), Algorithms and Complexity (AL), Programming Languages (PL) and Software Engineering (SE).

Systems

Courses that span Systems Fundamentals (SF), Architecture and Organization (AR), Operating Systems (OS), Parallel and Distributed Computing (PDC), Networking and Communication (NC), Security (SEC) and Data Management (DM).

Applications

Courses span Graphics and Interactive Techniques (GIT), Artificial Intelligence (AI), Specialized Platform Development (SPD), Human-Computer Interaction (HCI), Security (SEC) and Data Management (DM).

Introduction to Data Science:

- GIT-Visualization (8 hours): types of visualization, libraries, foundations
- GIT-SEP (2 hours): ethically responsible visualization
- DM-Core: Parallel and distributed processing (MapReduce, cloud frameworks, etc.)
- DM-Modeling: Graph representations, entity resolution
- DM-Querying (4 hours): SQL, query formation
- DM-NoSQL (2 hours): Graph DBs, data lakes, data consistency
- DM-Security: privacy, personally identifying information and its protection
- DM-Analytics (2 hours)
- DM-SEP (2 hours): Data provenance
- AI-ML (17 hours): Data preprocessing, missing data imputation, supervised/semi-supervised/unsupervised learning, text analysis, graph analysis and PageRank, experimental methodology, evaluation, and ethics
- AI-SEP (3 hours): Applications specific to data science, interspersed throughout the course
- MSF-Statistics: Statistical analysis, hypothesis testing, experimental design

Robotics to include the following:

- AI-Robo: Robotics: (25 hours)
- SPD-D: Robot Platforms: (4 hours) (focusing on hardware, constraints/considerations, and software architectures; some other topics in SPD/Robot Platforms overlap with AI/Robotics)
- AI-Search: Search: (4 hours) (selected topics well-integrated with robotics, e.g., A* and path search)
- AI-ML: (6 hours) (selected topics well-integrated with robotics, e.g., neural networks for object recognition)
- AI-SEP: (3 hours) (should be integrated throughout the course; robotics is already a huge application, so this really should focus on societal impact and specific robotic applications)

Prerequisites:

- CS2
- Linear algebra

Skill statement: A student who completes this course should be able to understand and use robotic techniques to perceive the world using sensors, localize the robot based on features and a map, and plan paths and navigate in the world in simple robot applications. They should understand and be able to apply simple computer vision, motion planning, and forward and inverse kinematics techniques.

Media Computation (CS I):

GIT-Foundations (4 hours)
GIT-Rendering (6 hours)
GIT-Interaction (3 hours)
SDF-Foundations (10 hours)
AL-Fundamentals (7 hours)
HCI-User (5 hours)
GIT-SEP (4 hours)

Facing the User

GIT-Foundations (4 hours)
GIT-Rendering (6 hours)
GIT-Interaction (3 hours)
HCI-User (8 hours)
HCI-Accessibility (3 hours)
HCI-SEP (4 hours)
SE-Testing (4 hours)
SPD-Web/SPD-Game/SPD-Mobile (8 hours)

Mobile Computing

- DM-Modeling (2 hours)
- DM-Querying (2 hours)
- SPD-Common: Overview of development platforms (3 hours)
- SPD-Common: Considerations and Requirements (4 hours)
- SPD-Mobile: Data Management (3 hours)
- SPD-Mobile: Development (6 hours)
- SPD-Mobile: Mobile Platform Constraints (4 hours)
- SPD-Mobile: Access (3 hours)

- SPD-Mobile: Architecture (3 hours)
- SPD-Mobile: Storage Solutions (4 hours)
- SPD-Mobile: Specification and testing (3 hours)
- SPD-Mobile: Asynchronous computing (3 hours)
- SPD-SEP/Mobile:

Curricular Packaging

8 Course Model

This is a minimal course configuration that covers all the CS core topics. But, it does not leave much room for exploration:

1. CS I (SDF, SEP)
2. CS II (SDF, AL, SEP, Generic KA)
3. Math and Statistical Foundations (MSF)
4. Algorithms (AL, PDC, SEP)
5. Introduction to Systems (SF, OS, AR, NC, SEP)
6. Formal Methods (FPL, AL-Formal, PDC, SEP)
7. Introduction to Applications (SEC, AI, HCI, GIT, SPD, DM, SEP)
8. Capstone (SE, SEP)

The capstone course is expected to provide the opportunity to cover any CS core topics not covered elsewhere in the curriculum.

12 Course Model

Please make sure CS core is covered no matter the choice of electives:

1. CS I (SDF, SEP)
2. CS II (SDF, AL, DM, SEP, Generic KA)
3. Math and Statistical Foundations (MSF)
4. Algorithms (AL, AI, SEC, SEP)
5. Introduction to Systems (SF, OS, AR, NC)
6. Programming Languages (FPL, AL, PDC, SEP)
7. Software Engineering (SE, HCI, GIT, PDC, SPD, DM, SEP)
8. Two from Systems electives:
    a. Operating Systems (OS, PDC)
    b. Computer Architecture (AR)
    c. Parallel and Distributed Computing (PDC)
    d. Networking (NC, SEC, SEP)
    e. Databases (DM, SEP)
9. Two electives from Applications:
    a. Artificial Intelligence (AI, SPD, SEP)
    b. Graphics (GIT, HCI, SEP)
    c. Application Security (SEC, SEP)
    d. Human-Centered Design (HCI, GIT, SEP)
10. Capstone (SE, SEP)

The capstone course is expected to provide the opportunity to cover any CS core topics not covered elsewhere in the curriculum.

16 Course Model

Three different models are offered here, each with its own benefits.

Model 1:

1. CS I (SDF, SEP)
2. CS II (SDF, AL, DM, SEP)
3. Math and Statistical Foundations (MSF)
4. Algorithms (AL, SEP)
5. Introduction to Systems (SF, SEP)
6. Programming Languages (FPL, AL, PDC, SEP)
7. Theory of Computation (AL, SEP)
8. Software Engineering (SE, HCI, GIT, PDC, SPD, DM, SEP)
9. Operating Systems (OS, PDC, SEP)
10. Computer Architecture (AR, SEP)
11. Parallel and Distributed Computing (PDC, SEP)
12. Networking (NC, SEP)
13. Pick one of:
    a. Introduction to Artificial Intelligence (AI, SEP)
    b. Machine Learning (AI, SEP)
    c. Robotics (AI, SPD, SEP)
14. Pick one of:
    a. Graphics (GIT, SEP)
    b. Human-Centered Design (GIT, SEP)
    c. Animation (GIT, SEP)
    d. Virtual Reality (GIT, SEP)
15. Security (SEC, SEP)
16. Capstone (SE, SEP)

Model 2:

1. CS I (SDF, SEP)
2. CS II (SDF, AL, DM, SEP)
3. Math and Statistical Foundations (MSF, AI, DM)
4. Algorithms (AL, SEP)
5. Introduction to Systems (SF, SEP)
6. Programming Languages (FPL, AL, PDC, SEP)
7. Theory of Computation (AL, SEP)
8. Software Engineering (SE, HCI, GIT, PDC, SPD, DM, SEP)
9. Operating Systems (OS, PDC, SEP)
10. Two electives from:
    a. Computer Architecture (AR, SEP)
    b. Parallel and Distributed Computing (PDC, SEP)
    c. Networking (NC, SEP)

    d. Network Security (NC, SEC, SEP)
    e. Security (SEC, SEP)
11. Pick three of:
    a. Introduction to Artificial Intelligence (AI, SEP)
    b. Machine Learning (AI, SEP)
    c. Deep Learning (AI, SEP)
    d. Robotics (AI, SPD, SEP)
    e. Data Science (AI, DM, GIT)
    f. Graphics (GIT, SEP)
    g. Human-computer interaction (HCI, SEP)
    h. Human-Centered Design (GIT, HCI, SEP)
    i. Animation (GIT, SEP)
    j. Virtual Reality (GIT, SEP)
    k. Physical Computing (GIT, SPD, SEP)
12. Society, Ethics and Professionalism (SEP)
13. Capstone (SE, SEP)

Model 3:

1. CS I (SDF, SEP)
2. CS II (SDF, AL, DM, SEP)
3. Math and Statistical Foundations (MSF)
4. Algorithms (AL, AI, SEC, SEP)
5. Introduction to Systems (SF, OS, AR, NC)
6. Programming Languages (FPL, AL, PDC, SEP)
7. Software Engineering (SE, HCI, GIT, PDC, SPD, DM, SEP)
8. Two from Systems electives:
    a. Operating Systems (OS, PDC)
    b. Computer Architecture (AR)
    c. Parallel and Distributed Computing (PDC)
    d. Networking (NC, SEC, SEP)
    e. Databases (DM, SEP)
9. Two electives from Applications:
    a. Artificial Intelligence (AI, SPD, SEP)
    b. Graphics (GIT, HCI, SEP)
    c. Application Security (SEC, SEP)
    d. Human-Centered Design (HCI, GIT, SEP)
10. Three open CS electives
11. Society, Ethics and Professionalism (SEP) course
12. Capstone (SE, SEP)

Section 3

A Competency Framework

A Competency Model Framework

Definition of Competency

Competency was defined as the sum of knowledge, skills and dispositions in IT2017 [7]. Dispositions are defined as *cultivable behaviors desirable in the workplace* [15].

Competency = Knowledge + Skills + Dispositions *in context*

In CC 2020 [8], competency was further elaborated as the sum of the three within the performance of a task. Instead of the additive model of IT 2017, CC2020 defined competency as an intersection of the three:

Competency = Knowledge ∩ Skills ∩ Dispositions

In CS2023, competency is treated as a point in a 3D space with knowledge, skills and dispositions as the three axes of the space (Figure 1) [15]: all three are required for proper execution of a task. Knowledge is covered by topics enumerated in knowledge units and knowledge areas; skills are identified as one or some of *Explain*, *Apply*, *Evaluate* and *Develop*. In the knowledge model (Section 2), appropriate dispositions were identified for each knowledge area.

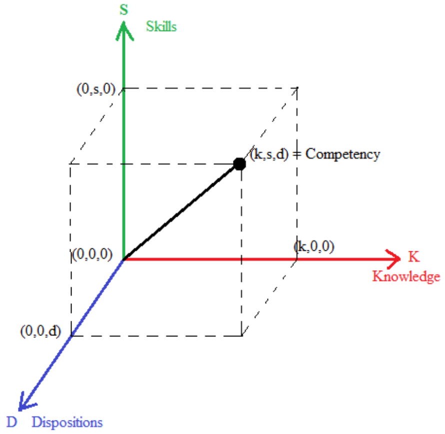


Figure 1. Competency as a point in a 3D space [15].

Competency is the application of knowledge, skills and dispositions in the performance of a task. For a given learner at a given moment and a given task, competency is represented as a point in this 3D space. For a set of tasks, the competency of a learner is a point cloud in this 3D space.

In the specification of a competency, the task is the sole independent variable. The knowledge, skills and dispositions needed to complete a task depend on the task and vary from one task to another. So, a competency model of a curriculum should necessarily start with identification of the targeted tasks. To this end, in this section:

1. A framework is proposed for systematically identify tasks and representative tasks are identified;
2. A format is introduced for competency specification;
3. Competency specifications are provided for selected tasks identified in step 1 using the format in step 2; and
4. An algorithm is provided for educators to build a competency model that is tailored to their local needs.

A Framework for Identifying Tasks

Computer science is a versatile discipline: the range of the tasks for which it prepares graduates is vast. In order to keep the task of identifying tasks tractable:

- The list will be restricted to atomic tasks that can be combined in infinite ways to create compound tasks to suit local needs;
- Instead of exhaustively listing all the tasks, a framework will be proposed for systematically identifying atomic tasks.

The framework for systematically identifying atomic tasks consists of three dimensions: component, activity and constraint. In a task statement, the component is typically the noun, the activity the verb and the constraint either an adjective or adverb.

The framework is tailored to the three competency areas proposed in the knowledge model:

- **Software:** Specifications that span Software Development Fundamentals (SDF), Algorithmic Foundations (AL), Foundations of Programming Languages (FPL) and Software Engineering (SE).
- **Systems:** Specifications that span Systems Fundamentals (SF), Architecture and Organization (AR), Operating Systems (OS), Parallel and Distributed Computing (PDC), Networking and Communication (NC), Security (SEC) and Data Management (DM).
- **Applications:** Specifications span Graphics and Interactive Techniques (GIT), Artificial Intelligence (AI), Specialized Platform Development (SPD), Human-Computer Interaction (HCI), Security (SEC) and Data Management (DM).

The following is an initial list of components in these three competency areas:

- Software: Program, algorithm, and language/paradigm.
- Systems: Processor, storage, communication, architecture, I/O, data, and service.
- Applications: Input, computation, output and platform.

A representative set of activities applicable to the competency areas are design, develop, document, evaluate, maintain, improve, humanize and research. While most of the activities are self-explanatory, humanize refers to activities that address society, ethics and professionalism issues and research refers to activities that study theoretical underpinnings.

Constraints are categorized as follows:

- Problem constraints, e.g., task size (small versus large), problem (well-defined versus open-ended) and task agent (solo versus in a team);
- Solution constraints - Non-functional requirements such as efficiency, reliability, security, performance and scalability;
- Implementation constraints, e.g., parallel, distributed and virtualized.

The components, activities and constraints listed above are representative, not prescriptive or comprehensive. All three axes use nominal scale, with no ordinality implied.

Each atomic task is a point in the three-dimensional space of component x activity x constraint as shown in Figure 1. At the bottom-right of the figure are the following three tasks mapped on software competency area:

- Develop a program for an open-ended problem (blue star);
- Evaluate the efficiency of a parallel algorithm (green star);

- Research language features for writing secure code (red star).

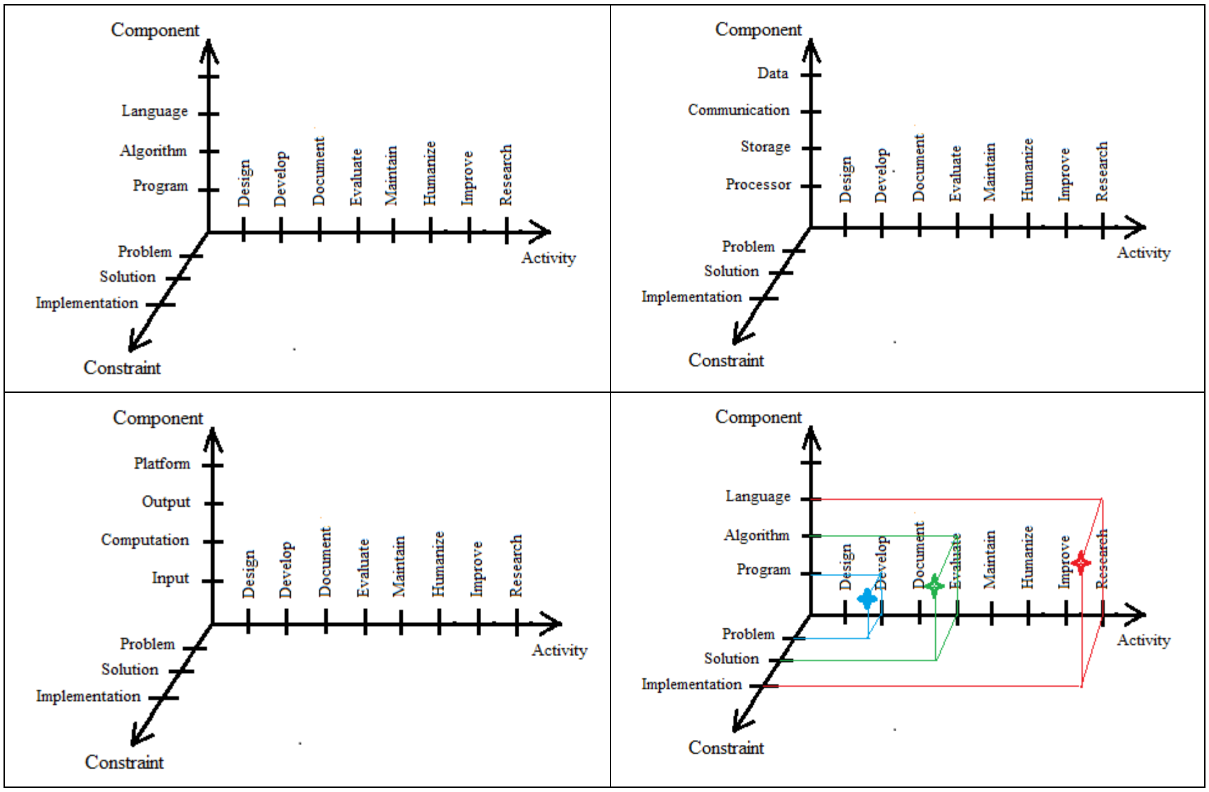


Figure 1: Software competency area (top-left); Systems competency area (top-right); Applications competency area (bottom-left) and three tasks mapped on software competency area (bottom-right)

The framework is offered as a starting point for identifying atomic tasks – it is meant to be used to **generate** tasks. One may want to add other components, activities and constraints to the framework as appropriate for their local needs. It is expected that most competency specifications will be written for compound tasks created by combining two or more atomic tasks, e.g., “Design, implement and document a parallelized scheduling program.”

Representative Tasks

Most of the representative tasks listed in this section are atomic in nature. The tasks are not restricted to CS or KA core topics only.

Software Competency Area

Component	Activity	Tasks
Program	Design	● Design an efficient algorithm for a problem ● Design efficient data structures for a problem

		● Design test cases to determine if a program is functionally correct ● Identify appropriate tools to assist in developing a program. ● Design an API for a service
Program	Develop	● Write a program for a given problem. ● Develop a program that leverages libraries and APIs. ● Automate testing of new code under development. ● Work in a team effectively to solve a problem.
Program	Document	● Document a program. ● Consistently format source code.
Program	Evaluate	• Evaluate an existing application (open source or proprietary) as a whole or partial solution for meeting a defined requirement
Program	Maintain	● Refactor a program. ● Perform code review to evaluate the quality of code
Program	Humanize	● Defend the design/choices made for a program. ● Ensure fair and equitable access in a program ● Document the accountability, responsibility and liability an individual/company assumes when releasing a given service/software/product ● Develop a strategy for a team to keep up to date with ethical, legal, and professional issues in relation to company strategy ● Incorporate legal and ethical privacy requirements into a given service/software/product's development cycle ● Convey the benefits of diverse development teams and user bases on company culture and the services/software/products the company provides, as well as the impacts that a lack of diversity can have on these
Program	Improve	● Debug a program
Program	Research	• Compute the running time of a program • Formally prove the correctness of code
Algorithm	Design	● Design an algorithm for a problem
Algorithm	Develop	● Implement an algorithm for a problem
Algorithm	Document	● Justify/defend an algorithm for a problem with a set of requirements
Algorithm	Evaluate	● Evaluate the efficiency of an algorithm
Algorithm	Maintain	● Redesign an algorithm to improve some non-functional characteristic (e.g., efficiency, reliability, security).
Algorithm	Humanize	● Justify that an algorithm is according to a human-dimension specification.
Algorithm	Improve	● Extend an algorithm for another similar problem
Algorithm	Research	• Prove the correctness of an algorithm • Compute the run time of an algorithm
Language / Paradigm	Design	● Select a language/paradigm for an application.
Language / Paradigm	Develop	● Effectively use the type system of a language to develop safe and secure software.

		● Write a program using multiple languages and have the components interact effectively and efficiently with each other.
Language / Paradigm	Document	● Justify the choice of a paradigm/language for a program. ● Write a white paper to describe how a program is translated into machine code and executed. ● White a white paper explaining how a program executes in an efficient manner with respect to memory and CPU utilization.
Language / Paradigm	Evaluate	● Evaluate the appropriateness of a language/paradigm for an application. ● Explain the benefits and challenges of converting an application into parallel/distributed version. ● White a white paper explaining how a program effectively utilizes language features to make it safe and secure
Language / Paradigm	Maintain	
Language / Paradigm	Humanize	
Language / Paradigm	Improve	
Language / Paradigm	Research	

Systems Competency Area

Component	Activity	Tasks
Processor	Design	● Revise a specification to enable parallelism and distribution without violating other essential properties or features ● Choose an application-specific scheduling algorithm
Processor	Develop	● Implement an application-specific scheduling algorithm ● Develop a version of your CPU-based application to run on a hardware accelerator (GPU, TPU, NPU). ● Implement a parallel/distributed component based on a known algorithm
Processor	Document	
Processor	Evaluate	● Describe and compare different scheduling algorithms ● Evaluate the performance-watt of your machine learning model deployed on an embedded device
Processor	Maintain	
Processor	Humanize	
Processor	Improve	● Improve the performance of a sequential application or component by introducing parallelism and/or distribution ● Identify and repair a performance problem due to sequential bottlenecks
Processor	Research	
Storage	Design	
Storage	Develop	
Storage	Document	

Storage	Evaluate	● Assess the performance implications of cache memories in your application ● Apply knowledge of operating systems to assess page faults in CPU-GPU memory management and their performance impact on the accelerated application.
Storage	Maintain	
Storage	Humanize	●
Storage	Improve	●
Storage	Research	●
I/O	Design	● Design software modules for sensor hardware integration.
I/O	Develop	● Develop a sensing-actuator robotics arm for an automated manufacturing cell. ● Develop a benchmarking software tool to assess the performance gain in removing I/O bottlenecks in your code.
I/O	Document	
I/O	Evaluate	
I/O	Maintain	
I/O	Humanize	
I/O	Improve	
I/O	Research	
Communication	Design	● Design a networking protocol. ● Design software that enables safe communication between processes
Communication	Develop	● Develop a model to abstract a networked environment. ● Develop a networked application ● Deploy and securely operate a network of wireless sensors. ● Develop software that enables safe communication between processes
Communication	Document	
Communication	Evaluate	● Evaluate multiple network architectures and network elements to meet needs. ● Evaluate the performance of a network, in specific latency, throughput, congestion, and various service levels.
Communication	Maintain	● Defend the network from an ongoing distributed denial-of-service attack.
Communication	Humanize	● Write a white paper to explain stakeholder needs of a given networked environment. ● Write a white paper to explain social, ethical, and professional issues governing the design and deployment of networked systems.
Communication	Improve	● Identify gray failures in a datacenter network. ● Identify and repair a performance problem due to communication or data latency
Communication	Research	
Architecture	Design	● Describe common criteria of system design

		● Design a system to meet functional/non-functional specifications. ● Document a system’s design choices and proposed system hardware and software architecture.
Architecture	Develop	● Deploy a system in a cloud environment ● Deploy an application component on a virtualized container
Architecture	Document	
Architecture	Evaluate	● Evaluate the performance of a given system ● Find the performance bottleneck of a given system ● Choose among different parallel/distributed designs for components of a given system
Architecture	Maintain	
Architecture	Humanize	
Architecture	Improve	● Find and fix bugs in a system
Architecture	Research	
Data	Design	● Determine how to store a new application’s data.
Data	Develop	● Access data from a database for some purpose. ● Create a database for a new application.
Data	Document	
Data	Evaluate	
Data	Maintain	● Get back online after a disruption (e.g. power outage)
Data	Humanize	● Produce a white paper assessing the social and ethical implications for collecting and storing the data from a new (or existing) application. (risk) ● Assess the legal and ethical implications of collecting and using customer/user data ● Write a white paper on data privacy implications in developing a data collection application, e.g., medical device for continuous patient sensor data collection.
Data	Improve	● Improve a database application’s performance (speed) ● Secure data from unauthorized access ● Modify a concurrent system to use a more scalable, reliable or available data store
Data	Research	
	Design	
	Develop	
	Document	
	Evaluate	
	Maintain	
	Humanize	
	Improve	
	Research	

- Identify and repair a performance problem due to resource management overhead

- PDC10: Identify and repair a reliability or availability problem

Applications Competency Area

Component	Activity	Tasks
Input	Design	• Design a user interface for an application
Input	Develop	• Implement the user interface of an application
Input	Document	
Input	Evaluate	
Input	Maintain	
Input	Humanize	• Write a paper on the accessibility of a user interface
Input	Improve	
Input	Research	
Computation	Design	• Specify the operators and partial-order planning graph to solve a logistics problem, showing all ordering constraints.
Computation	Develop	• Implement an agent to play a two-player complete information board game. • Implement an agent to play a two-player incomplete information board game. • Write a program that uses Bayes rule to predict the probability of disease given the conditional probability table and a set of observations. • Train and evaluate a neural network for playing a video game (e.g., Mario, Atari). • Develop a tool for identifying the sentiment of social media posts. • Write a program to solve a puzzle or gridworld
Computation	Document	
Computation	Evaluate	• Compare the performance of three supervised learning models on a dataset • Explain some of the pitfalls of deep generative models for image or text and how this can affect their use in an application.
Computation	Maintain	
Computation	Humanize	• Write an essay on the effects of data set bias and how to mitigate them.
Computation	Improve	
Computation	Research	
Output	Design	
Output	Develop	
Output	Document	
Output	Evaluate	
Output	Maintain	
Output	Humanize	
Output	Improve	
Output	Research	

Platform	Design	• Determine whether to develop an app natively or using cross-platform tools.
Platform	Develop	• Create a mobile app that provides a consistent user experience across various devices, screen sizes, and operating systems. • Build a secure web page for evolving business needs • Develop application programming interfaces (APIs) to support mobile functionality.
Platform	Document	
Platform	Evaluate	• Analyze people's experience using a novel peripheral for an immersive system, with attention to usability and accessibility specifications
Platform	Maintain	
Platform	Humanize	
Platform	Improve	• Optimize a secure web page for evolving business needs
Platform	Research	
	Design	
	Develop	
	Document	
	Evaluate	
	Maintain	
	Humanize	
	Improve	
	Research	

A Format for Competency Specification

The following format will be used for **competency specifications**:

- **Task:** *What* an employer might want done in the context of a job.
- **Competency statement:** *What* a graduate might bring to bear in terms of knowledge and skills to attempt the task - this is based on learning outcomes in the curriculum.
- **Competency area:** Software/Systems/Applications/Other
- **Competency unit/activity:** Design/Develop/Document/Evaluate/Maintain/Humanize/Improve/Research/…
- **Required knowledge areas and knowledge units:** The knowledge units and the knowledge areas of which they are a part needed to complete the task.
- **Required skill level:** Explain/Apply/Evaluate/Develop
- **Desirable professional dispositions:** Adaptable/Collaborative/Inventive/Meticulous/Persistent/Proactive/Responsive/Self-Directed

This format differs from earlier proposals (e.g., [8, 16, 17]) in some key respects:

- The task is separated from the competency statement, since the task is the independent component of the specification, with all the other components depending on it. The task is typically written in layman terms whereas technical details for completing the task are included in the competency statement.
- Dispositions are listed in knowledge areas and not in competency specifications. A competency specification inherits its dispositions from the knowledge areas listed in it.

The reader is invited to adopt/adapt the format that best meets their local needs and suits their preferences.

Sample Competency Specifications

The following are some sample competency specifications that draw upon various knowledge areas of computer science. They illustrate a range of competencies across all three competency areas (Software, Systems and Applications), multiple competency units/activities (Design, Develop, Document, Evaluate, Maintain, Humanize, Improve, Research) and all four skill levels (Explain, Apply, Evaluate and Develop) at the undergraduate level. Some draw upon a single knowledge area while others span multiple knowledge areas.

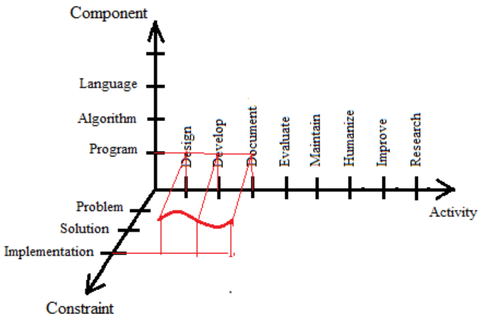
Software Competency Area

- **Task SDF1:** Develop test cases to determine if a program is functionally correct.
- **Competency statement:** Develop test cases and test a given program.
- **Competency area:** Software
- **Competency unit/activity:** Develop
- **Required knowledge areas and knowledge units:**
    - SDF-Software Develop Practices
- **Required skill level:** Develop
- **Desirable professional dispositions:**

- **Task SDF2:** Perform code review to evaluate the quality of code.
- **Competency statement:** Read and understand the code and identify errors in it.
- **Competency area:** Software, Application
- **Competency unit/activity:** Document, Evaluate, Improve
- **Required knowledge areas and knowledge units:**
    - SDF-Fundamental Programming Concepts, Software Develop Practices
- **Required skill level:** Evaluate
- **Desirable professional dispositions:**

- **Task SE1:** Automate testing of new code under development.
- **Competency statement:** Break down the intended behavior of an API into one or more relevant inputs that can be used to ensure that behavior remains functional over time.
- **Competency area:** Software, Application

- **Competency unit/activity:** Design, Develop, Document
- **Required knowledge areas and knowledge units:**
    - SDF-Development Methods
    - SE-Software Construction
- **Required skill level:** Apply, Develop
- **Desirable professional dispositions:**



- **Task FPL1:** Make an informed decision regarding which programming language/paradigm to select and use for a specific application.
- **Competency statement:** Apply knowledge of multiple programming paradigms, including their strengths and weaknesses relative to the application to be developed, and select an appropriate paradigm and programming language.
- **Competency area:** Software, Application
- **Competency unit/activity:** Design, Develop, Evaluate
- **Required knowledge areas and knowledge units:**
    - FPL-Object-Oriented Programming
    - FPL-Functional Programming
    - FPL-Logic Programming
    - FPL-Event-Driven and Reactive programming
    - FPL-Type Systems
    - FPL-Language Translation and Execution
    - FPL-Language Pragmatics
    - FPL-Embedded Computing and Hardware Interface
    - FPL-Advanced Programming Constructs
- **Required skill level:** Explain, Evaluate
- **Desirable professional dispositions:**

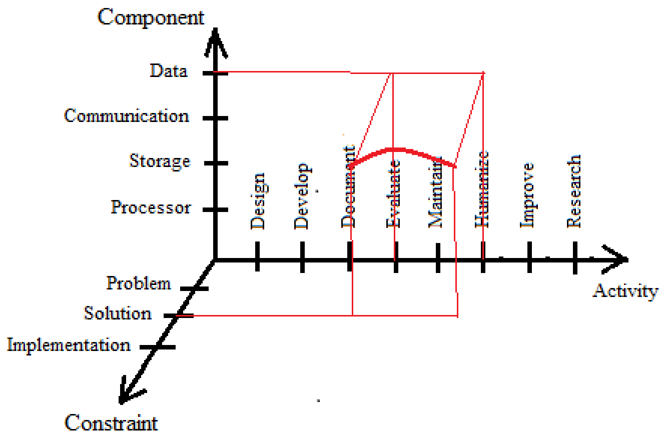
- **Task FPL2:** Effectively use a programming language's type system to develop safe and secure software.
- **Competency statement:** Apply knowledge of static and dynamic type rules for a language to ensure an application is safe, secure, and correct.
- **Competency area:** Software, Application

- **Competency unit/activity:** Develop
- **Required knowledge areas and knowledge units:**
    - FPL-Type Systems
- **Required skill level:** Develop
- **Desirable professional dispositions:**

Systems Competency Area

- **Task AR1:** Develop a version of a CPU-based application to run on a hardware accelerator (GPU, TPU, NPU).
- **Competency statement:** Apply knowledge from systems design to accelerate an application code and evaluate the code speed-up.
- **Competency area:** Software, Systems
- **Competency unit/activity:** Design, Develop, Document
- **Required knowledge areas and knowledge units:**
    - AR-Heterogeneous Architectures
    - PD-Parallel Architecture
    - SF-System Design
- **Required skill level:** Evaluate, Develop
- **Desirable professional dispositions:**

- **Task SEP1:** Produce a white paper assessing the social and ethical implications of collecting and storing the data from a new (or existing) application.
- **Competency statement:** Identify the stakeholders and evaluate the potential long-term consequences for the collection and retention of data objects. Consider both potential harm from unintended data use and from data breaches.
- **Competency area:** Systems
- **Competency unit/activity:** Evaluate, Humanize.
- **Required knowledge areas and knowledge units:**
    - SEP-Social Context
    - SEP-Methods for Ethical Analysis
    - SEP-Privacy and Civil Liberties
    - SEP-Professional Ethics
    - SEP-Security Policies, Laws and Computer Crimes
    - SEP-Equity, Diversity and Inclusion
    - DM-The Role of Data
    - SEC-Foundational Security
- **Required skill level:** Evaluate, Explain
- **Desirable professional dispositions:**



- **Task DM1:** Secure data from unauthorized access.
- **Competency statement:** Create database views to ensure data access is appropriately limited.
- **Competency area:** Systems
- **Competency unit/activity:** Maintain
- **Required knowledge areas and knowledge units:**
  - DM-The Role of Data
  - DM-Relational Databases
  - DM-Query Processing
  - SEP-Security Policies, Laws and Computer Crimes
  - SEP-Professional Ethics
  - SEP-Privacy and Civil Liberties
  - SEC-Foundational Security
- **Required skill level:** Develop
- **Desirable professional dispositions:**

- **Task DM2:** Create a database for a new application.
- **Competency statement:** Design the data storage needs (data modeling), assess the social and ethical implications for collecting and storing the data, determine how to store a new application's data (RDBMS vs NoSQL), and create the database, including appropriate indices.
- **Competency area:** Software, Systems
- **Competency unit/activity:** Design, Develop
- **Required knowledge areas and knowledge units:**
  - DM-The Role of Data
  - DM-Core Database Systems Concepts
  - DM-Data Modeling
  - DM-Relational Databases
  - DM-NoSQL Systems

  - DM-DBMS Internals
  - SEP-Social Context
  - SEP-Methods for Ethical Analysis
  - SEP-Privacy and Civil Liberties
  - SEP-Professional Ethics
  - SEP-Security Policies, Laws and Computer Crimes
  - SEP-Equity, Diversity and Inclusion
  - SEC-Foundational Security
- **Required skill level:** Develop
- **Desirable professional dispositions:**

- **Task NC1:** Develop networked application.
- **Competency statement:** Design and implement networked applications that satisfy specific requirements, including various constraints during usage.
- **Competency area:** Software, Systems
- **Competency unit/activity:** Design, Develop
- **Required knowledge areas and knowledge units:**
  - NC-Networked Applications
  - NC-Security
  - NC-Mobility
- **Required skill level:** Develop
- **Desirable professional dispositions:**

- **Task NC2:** Evaluate the performance of a network, in specific latency, throughput, congestion, and various service levels.
- **Competency statement:** Identify and evaluate various indicators of the performance of a network to suit specific needs.
- **Competency area:** Systems
- **Competency unit/activity:** Evaluate
- **Required knowledge areas and knowledge units:**
  - NC-Networked Applications
  - NC-Routing and Forwarding
- **Required skill level:** Evaluate
- **Desirable professional dispositions:**

- **Task OS1:** Deploy an application component on an operating system runtime/virtualized operating system/container.
- **Competency statement:** Identify and mitigate potential problem with deployment; automate setup of deployment environment; set up monitoring of component execution.
- **Competency area:** Systems

- **Competency unit/activity:** Evaluate, Maintain, Improve
- **Required knowledge areas and knowledge units:**
  - OS-Role and Purpose of Operating Systems
  - OS-Principles of Operating Systems
  - OS-Concurrency
  - OS-Scheduling
  - OS-Process Model
  - OS-Memory Management
  - OS-Protect and Safety
  - AR-Assembly Level Machine Organization
- **Required skill level:** Apply
- **Desirable professional dispositions:**

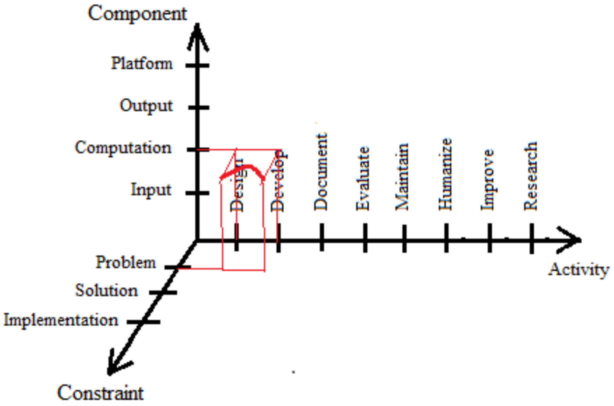
- **Task PDC1:** Improve the performance of a sequential application or component by introducing parallelism and/or distribution.
- **Competency statement:** Evaluate how and when parallelism and/or distribution can improve (or not improve) performance well enough to identify opportunities, as well as implement them and measure results.
- **Competency area:** Software, Systems, Application
- **Competency unit/activity:** Design, Develop, Evaluate, Improve
- **Required knowledge areas and knowledge units:**
  - PDC-Software Engineering
  - PL-Parallel and Distributed Computing
- **Required skill level:** Evaluate, Develop
- **Desirable professional dispositions:**

- **Task SF1:** Find the performance bottleneck of a given system.
- **Competency statement:** Given a system and its target deployment environment, find its performance bottleneck (e.g., memory, CPU, and networking) through analytical derivation or experimental study.
- **Competency area:** Systems
- **Competency unit/activity:** Design, Document, Evaluate
- **Required knowledge areas and knowledge units:**
  - SF-System Performance
  - SF-Performance Evaluate
  - SF-System Design
  - SF-Overview of Computer Systems
- **Required skill level:** Apply, Evaluate, Develop
- **Desirable professional dispositions:**

Applications Competency Area

- **Task SE2:** Work on a team effectively.
- **Competency statement:** Focus on long-term team dynamics and communicate effectively.
- **Competency area:** Applications
- **Competency unit/activity:** Document, Humanize, Improve
- **Required knowledge areas and knowledge units:**
  - SE-Teamwork
- **Required skill level:** Apply
- **Desirable professional dispositions:**

- **Task AI1:** Implement an agent to make strategic decisions in a two-player adversarial game with uncertain actions (e.g., a board game, strategic stock purchasing).
- **Competency statement:** Use minimax with alpha-beta pruning, and possible chance nodes (expectiminimax), and heuristic move evaluation (at a particular depth) to solve a two-player zero-sum game.
- **Competency area:** Applications
- **Competency unit/activity:** Design, Develop
- **Required knowledge areas and knowledge units:**
  - AI-Search
  - AI-Fundamental Knowledge Representation and Reasoning
- **Required skill level:** Apply, Develop
- **Desirable professional dispositions:**



- **Task AI2:** Analyze tabular data (e.g., customer purchases) to identify trends and predict variables of interest.
- **Competency statement:** Use machine learning libraries, data preprocessing, training infrastructures, and evaluation methodologies to create a basic supervised learning pipeline.
- **Competency area:** Applications

- **Competency unit/activity:** Design, Develop, Evaluate
- **Required knowledge areas and knowledge units:**
    - AI-Machine Learning
    - AI-Applications and Societal Impact
- **Required skill level:** Apply, Develop
- **Desirable professional dispositions:**

- **Task AI3:** Critique a deployed machine learning model in terms of potential bias and correct the issues.
- **Competency statement:** Understand, recognize, and evaluate issues of data set bias in AI, the types of bias, and algorithmic strategies for mitigation.
- **Competency area:** Applications
- **Competency unit/activity:** Document
- **Required knowledge areas and knowledge units:**
    - AI-Machine Learning
    - AI-Applications and Societal Impact
- **Required skill level:** Explain
- **Desirable professional dispositions:**

- **Task GIT1:** Visualize a region's temperature.
- **Competency statement:** Given weather data, design and implement an animation depicting temperature changes for a region over time.
- **Competency area:** Applications
- **Competency unit/activity:** Design, Develop, Evaluate, Humanize
- **Required knowledge areas and knowledge units:**
    - GIT-Fundamental Concepts
    - GIT-Basic Rendering
    - GIT-Visualization
    - HCI-System Design
    - HCI-Understanding the User
- **Required skill level:** Apply, Evaluate, Develop
- **Desirable professional dispositions:**

- **Task HCI1:** Evaluate and provide recommendations to improve a user-facing system's usability, accessibility, and inclusivity.
- **Competency statement:** Demonstrate knowledge of various lenses for evaluating user-facing systems.
- **Competency area:** Applications
- **Competency unit/activity:** Evaluate, Document, Humanize, Improve
- **Required knowledge areas and knowledge units:**
    - HCI-User

    - HCI-Accessibility
    - HCI-Evaluation
    - HCI-Design
    - HCI-SEP
- **Required skill level:** Evaluate
- **Desirable professional dispositions:**

- **Task HCI2:** Determine what aspects of an implementation require revision to support internationalization.
- **Competency statement:** Evaluate a system to identify culturally-relevant or language-relevant text, symbols, and patterns that may vary by locale.
- **Competency area:** Software, Systems, Applications
- **Competency unit/activity:** Design, Evaluate, Humanize, Improve
- **Required knowledge areas and knowledge units:**
    - HCI-User
    - HCI-Accountability
    - HCI-Accessibility
    - HCI-Evaluation
- **Required skill level:** Evaluate
- **Desirable professional dispositions:**

- **Task HCI3:** Prototype a user-interface based on a design specification and industry-standard guidelines.
- **Competency statement:** Develops a prototype for a user-facing interface of a software system according to a provided software design specification and in line with professional standards and expectations.
- **Competency area:** Software, Applications
- **Competency unit/activity:** Design, Develop
- **Required knowledge areas and knowledge units:**
    - HCI-User
    - HCI-Accountability
    - HCI-Accessibility
- **Required skill level:** Develop
- **Desirable professional dispositions:**

- **Task SEP1:** Assess the ethical and societal implications of deploying a given AI-powered service/software/product.
- **Competency statement:** Determine who will be affected and how.
- **Competency area:** Applications
- **Competency unit/activity:** Evaluate, Humanize
- **Required knowledge areas and knowledge units:**
    - AI-Fundamental Issues
    - SEP-Social Context

  - SEP-Methods for Ethical Analysis
  - SEP-Privacy and Civil Liberties
  - SEP-Justice. Equity, Diversity and Inclusion
- **Required skill level:** Explain, Evaluate
- **Desirable professional dispositions:**

- **Task SEP2:** Document the accountability, responsibility and liability a company assumes when releasing a given service/software/product.
- **Competency statement:** Gather the appropriate information and present it in a coherent and useful manner.
- **Competency area:** Applications
- **Competency unit/activity:** Document
- **Required knowledge areas and knowledge units:**
  - SEP-Professional Ethics
  - SEP-Intellectual Property
  - SEP-Privacy and Civil Liberties
  - SEP-Professional Communication
  - SEP-Justice. Equity, Diversity and Inclusion
- **Required skill level:** Explain
- **Desirable professional dispositions:**

- **Task SEP3:** Demonstrate cultural awareness and cultural competence in written and verbal communication.
- **Competency statement:** Professionally communicate with required stakeholders with cultural sensitivity.
- **Competency area:** Applications
- **Competency unit/activity:** Document, Humanize
- **Required knowledge areas and knowledge units:**
  - SEP-Social Context
  - SEP-Professional Communication
  - SEP-Justice. Equity, Diversity and Inclusion
- **Required skill level:** Explain
- **Desirable professional dispositions:**

- **Task SPD1:** Determine whether to develop an app natively or using cross-platform tools.
- **Competency statement:** Understand performance and scalability issues, and evaluate different approaches and tools by carefully considering factors such as app requirements, target audience, time-to-market, and costs.
- **Competency area:** Applications
- **Competency unit/activity:** Evaluate

- **Required knowledge areas and knowledge units:**
    - SE-Tools and Environments
    - SPD-Common Aspects
    - SPD-Mobile Platform
- **Required skill level:** Explain
- **Desirable professional dispositions:**

- **Task SPD2:** Build and optimize a secure web page for evolving business needs using a variety of *appropriate* programming languages.
- **Competency statement:** Evaluate potential security hazards and apply optimization techniques.
- **Competency area:** Applications
- **Competency unit/activity:** Evaluate
- **Required knowledge areas and knowledge units:**
    - AR-Performance and Energy Efficiency
    - NC-Network Security
    - OS-Protection and Safety
    - SF-System Security
    - SE-Software Design
    - SE-Tools and Environments
    - SPD-Common Aspects
    - SPD-Mobile Platform
    - SEP-Privacy
- **Required skill level:** Develop
- **Desirable professional dispositions:**

Using the Competency Model Framework

The following is proposed as the procedure for creating a competency model for a curriculum, to be carried out in consultation with local stakeholders (academics, industry representatives, policy makers, etc.):

1. Identify the competency area(s) targeted by the curriculum based on local needs;
2. For each targeted competency area, identify the atomic tasks that must be targeted by the curriculum using the component x activity x constraint three-dimensional space of the competency area shown in Figure 1. The targeted atomic tasks will each be a point in the three-dimensional space as illustrated at the bottom-right in Figure 1.
3. If it is desirable to reduce the number of competency specifications, create compound tasks by combining two or more related atomic tasks.
4. Use a format of choice to write a competency specification for each atomic or compound task identified in the previous two steps.
5. The aggregate of the competency specifications for all the identified atomic/compound tasks is the competency model of the curriculum.

- Ensure that the competency model draws upon all the topics identified as CS core.

Knowledge Model or Competency Model?

Knowledge Model Versus Competency Model

A knowledge model organizes content into knowledge areas, which are silos of related content. Each knowledge area consists of multiple knowledge units, and each knowledge unit consists of multiple topics. This epistemological organization of content facilitates the process of designing courses and curricula: multiple courses may be carved out of a single knowledge area and a course may draw content from multiple knowledge areas. Therefore, a knowledge model with its initial emphasis on knowledge areas facilitates the needs of *teaching*.

A competency model consists of competency specifications, with each specification consisting of a vernacular description called the competency statement and enumeration of the knowledge, skills and dispositions needed to complete the task described in the competency statement [16, 17]. By grouping content needed for each competency specification, a competency model helps learners make associations among complementary concepts from multiple knowledge areas. By explicitly listing the tasks a graduate should be expected to complete, it also facilitates evaluation of student learning.

A knowledge model with its initial emphasis on content and a competency model with its initial emphasis on outcomes are complementary views of the same learning continuum. For computer science, neither model is a substitute for the other. The two models complement each other, and work better considered together than apart. So, a Combined Knowledge and Competency (CKC) model [15] will be used in this report that synergistically combines the two and offers the benefits of both.

CKC Model

The Combined Knowledge and Competency (CKC) model is illustrated in Figure 1 [15]. In the figure:

- The knowledge model appears on the left and consists of knowledge areas that in turn consist of knowledge units.
- The topics in the knowledge units are categorized as CS core (topics that every computer science graduate must know), KA core (recommended topics for inclusion in a dedicated coverage of a knowledge area) and Non-core (electives).
- A computer science program may choose to cover some knowledge areas in greater depth/breadth than other knowledge areas. When coherently chosen, the knowledge areas covered by a computer science program will constitute the program's competency area(s).
- The competency model appears on the right and consists of competency areas that in turn consist of competency units.
- Competency units are activities that apply to every competency area, such as: design, develop, document, evaluate, maintain, humanize, improve and theorize. A competency area is the sum of its competency units/activities. Whereas the number of competency areas targeted by a program indicates its breadth, the number competency units targeted by the program in each competency area indicates its depth.

- A competency is the application of knowledge, skills and dispositions to the completion of a task. Since task is the only objective component of a competency statement, tasks are separated out of competency statements and identified at the atomic level for each competency area.
- Even though the emphasis on dispositions is greater in a competency model, dispositions are generic to knowledge areas. So, they are associated with knowledge areas. This makes it easier for educators to consistently promote them during the accomplishment of tasks associated with the knowledge areas.
- Finally, skill levels connect tasks in the competency model with knowledge and dispositions in the knowledge model.

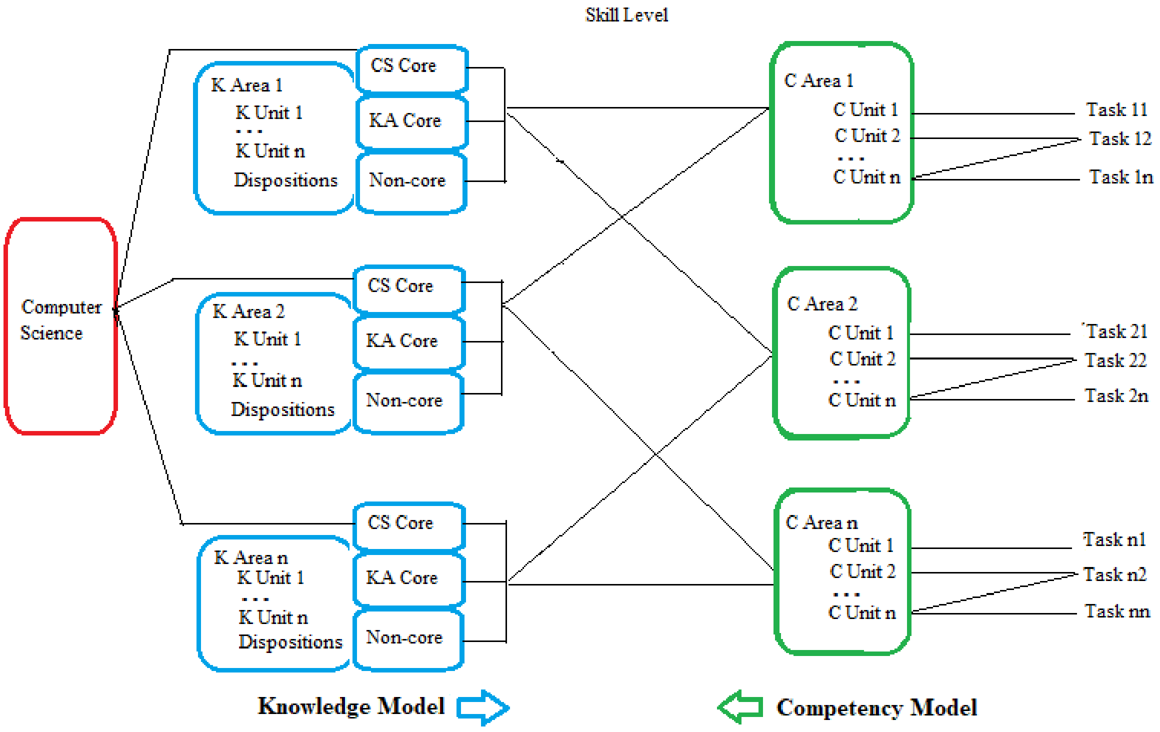


Figure 1. Combined Knowledge and Competency (CKC) Model of Computer Science Curricula.

To summarize the CKC model, competency areas are referred to in the knowledge model, knowledge areas are referred to in the competency model, skill levels provide alignment between the two models and dispositions are associated with knowledge areas in the knowledge model, but used to facilitate completion of tasks specified in the competency model. The knowledge component of the CKC model is presented in section 2. A framework for designing a competency model is presented earlier in section 3.

How to use the CKC Model

The following is proposed as the procedure for creating the curriculum of a computer science program from the CKC model [15]:

1. Identify the competency area(s) targeted by the curriculum based on local needs;

2. Design courses and curricula using the knowledge areas and knowledge units of the CKC model as described in Section 2;
3. Design a competency model consisting of competency specifications for the targeted competency area(s) as described in this section;
4. Use the courses and curricula designed in step 2 for instruction and the competency model designed in step 3 to evaluate the outcomes of the program.
5. In a cycle of continual improvement, repeat steps 1 – 4 to improve courses, competency statements and outcomes of the program.

Section 4

Curricular Issues

Characteristics of Graduates

Some possible characteristics are:

- Algorithmic problem-solver - Good solutions to common problems at an appropriate level of abstraction
- Competent programmer
- In possession of mental model of computation - deep learning
- Life-long learner
- Collaborative
- Socially responsible - ethical behavior
- Global and cultural competence
- Cross-disciplinary - understanding of non-computing disciplines
- Adversarial thinker/Computational thinker
- Think at multiple levels of abstraction.
- Adaptable
- Handle ambiguity and uncertainty
- Analytical and problem-solving skills
- Knowledge of algorithms and data structures
- Familiarity with software engineering principles
- Strong mathematical and logical skills
- Effective communication skills

We will endeavor to cite other articles where characteristics of CS graduates have been enumerated.

The Process

CS 2013 → Version Beta SC survey → Open feedback form (61 responses) → Article

Institutional Challenges

Some institutional challenges are:

- Faculty recruitment and retention
- Workload to manage explosive enrollments
- AI generation of code and its implications
- Integrating AI into the core educational requirements
- Fragmentation (Data Science, AI, etc. etc.) are all becoming degrees in their own right
- Rethinking core curriculum
- Unprepared high school students - Getting students to put the effort in to be successful as CS students.
- Elevating teaching track to professional status
- Certificates, micro-credentials, associates in computing
- CS for all (packaging it in an accessible way to a broad set of majors)
- Broadening Participation of under-represented groups in Computing
- Online education

The Process

CS 2013 → Version Beta SC survey → Open feedback form (32 responses) → Article

Generative AI and the Curriculum

Introduction

Generative AI technologies have the potential to greatly disrupt computer science education. While it is too early to confidently prognosticate how they will change computer science education, it is instructive to consider some of the ramifications already apparent. In this section, a few of the ramifications of generative AI technology are explored by knowledge area.

Implications by Knowledge Area

Algorithmic Foundations (AL)

-

Architecture and Organization (AR)

- The evolution needed in computer architecture to better support generative AI technologies may itself become an important area of study in the future.

Artificial Intelligence (AI)

- Generative AI capabilities call into question the need for traditional approaches in areas such as natural language processing. Then again, combining symbolic AI approaches with generative AI techniques may provide benefits not yet envisioned. Artificial Intelligence as a field of study will be significantly impacted by the rise and evolution of generative AI techniques..

Data Management (DM)

- Structured Query Language (SQL) is dead. Generative AI will replace SQL. However, computer science students must be taught to validate the results returned by generative AI technologies.

Foundations of Programming Languages (FPL)

- The next paradigm in programming might be conversational programming that involves providing the right prompts to help generative AI produce and test desired programs.

Graphics and Interactive Techniques (GIT)

- Issues of ownership and copyright of images created using generative AI must be raised and discussed with students.

Human-Computer Interaction (HCI)

-

Mathematical and Statistical Foundations (MSF)

-

Networking and Communication (NC)

- Generative AI technologies may be increasingly used to verify whether networking requirements are ambiguous or complete. They may be used to automatically generate configuration scripts.

Operating Systems (OS)

- Generative AI may be used for help with deployment scripts and system optimization.

Parallel and Distributed Computing (PDC)

-

Security (SEC)

-

Society, Ethics and Professionalism (SEP)

-

Software Development Fundamentals (SDF)

- While generative AI can be used to write simple programs, the responsibility to verify the correctness of the programs still falls on the user. So, even if for verification purposes only, computer science students still need to learn how to write programs. In order to provide appropriate prompts to generative AI, they need to be able to design and plan larger programs. How students design and write programs will evolve as generative AI technologies improve. How computer science educators must adapt to teaching programming with the help of generative AI is currently an open question.

Software Engineering (SE)

- LLMs are expected to have substantial impact on several aspects of the software process, including (but not limited to) development of new code, comprehension of complex logging and debugging artifacts, static analysis, and code reviewing. The most understandable and visible change is likely to be in development of new code: assistance technologies like GitHub's Copilot and other advanced auto-complete mechanisms can meaningfully improve development time, to a point. As most generative AI systems have adopted a similar UX, we should also expect that the IDE may stop being the primary environment for new development, and we anticipate a prompt-driven approach. In the ideal form, those prompts are a semi-formal specification of programmer intent: what type of API is the programmer attempting to create? If our LLMs proceed by generating an API and surfacing appropriate unit tests for that, the resulting workflow of Prompt → Generate API → Generate Tests → Generate Implementation may be a powerful up-lift, and force developers to think more about what is being created rather than how to implement it. Critically, this requires a substantially deeper investment in design (especially the vocabulary of design) and code comprehension, while potentially decreasing the need for hands-on programming time. Similar advances in static analysis and code review are anticipated to have meaningful impact on code quality and clarity, while ideally reducing the impact of implicit bias by increasing consistency and quality of comments and diagnostics.

Specialized Platform Development (SPD)

Systems Fundamentals (SF)

- Providing system support for generative AI applications is expected to turn into a robust area of research and education.

Implications for the Curriculum

- It is clear that students must be taught how to correctly use generative AI technologies for coursework. The boundary between using generative AI as a resource and using it to plagiarize must be clarified. The limitations (e.g., hallucinations) of the technology must be adequately discussed, as also inherent biases that may have been baked into the technology by virtue of the data used to train them.
- Every new technology has redefined the boundary between tasks that can be mechanized and those that will need human participation. Generative AI is no different. Correctly identifying the boundary will be the challenge for computer science educators going forward.
- Generative AI may be used to facilitate undergraduate research – regardless of their technical abilities, students can now be asked to design the correct prompts to recreate the software reported in research publications before proceeding to validate the results reported in the publications.
- Students may use generative AI to summarize assigned readings, help explain gaps in their understanding of course material and fill in gaps in the presentations of the classes they missed.

Pedagogical Considerations

Introduction

What are some current trends in the teaching and learning of computer science? What are the controversies of the day in terms of the pedagogy of computer science education? In this section, a top few trends, controversies, and challenges have been listed for each knowledge area. These issues are expected to influence the future evolution of computer science curricula.

Considerations by Knowledge Area

Algorithmic Foundations (AL)

- Should computer science graduates be able to explain, at some level, current popular CS conceptual areas that are in the news? For example, Consensus algorithms with Blockchain as an example, SHA-246, Quantum Computing, Large language models (AI at least) Each of these topics have received less than 50% support from the reviewers of the AL area. If Algorithmic Foundations means the first courses, which the committee doesn't believe, but the reviewers seem to think so, then this makes sense, but where do these emerging topics land? If Quantum , for example, is a new algorithmic foundation concept, then it belongs in AL? If we don't have these emerging topics anywhere, then we might as well have copied 2013 and published it (okay a bit for-all).
- Should we still teach low-level algorithms?

Architecture and Organization (AR)

Text

Artificial Intelligence (AI)

- Should we still teach symbolic AI?

Data Management (DM)

- How to solve the conundrum that for the most part, students write code that either reads/writes to a file, or is interactive. Yet, in industry, the vast majority of data is obtained programmatically from a database. Shouldn't this be how our curricula be structured as well?
- SQL vs NoSQL databases?

Foundations of Programming Languages (FPL)

- Should scripting be a paradigm?

Graphics and Interactive Techniques (GIT)

Text

Human-Computer Interaction (HCI)

Text

Mathematical and Statistical Foundations (MSF)

Faculty and students alike have strong opinions about how much and what math should be included in the CS curriculum. Generally, faculty, who themselves have strong theoretical training, are typically concerned about poor student preparation and motivation to learn math, while students complain about not seeing applications and wonder what any of the math has to do with the software jobs they seek. Even amongst faculty, there is recurring debate on whether calculus should be required of computer science students, accompanied by legitimate concern about the impact of calculus failure rates on computer science students. Yet, at the same time, the discipline has itself undergone a significant mathematical change: machine learning, robotics, data science, and quantum computing all demand a different kind of math than is typically covered in a standard discrete structures course. The combination of changing mathematical demands and inadequate student preparation or motivation, in an environment of enrollment-driven strain on resources, has become a key challenge for CS departments.

Summary of recommendations

- ***Standardize the prerequisites to discrete math***. The faculty survey shows that institutional variation in discrete-math prerequisites distributes nearly evenly across algebra, precalculus and calculus, suggesting differing approaches to the mathematical maturity sought. Requiring *precalculus* appears to be a reasonable compromise, so that students come in with some degree of comfort with symbolic math and functions.
- ***Include applications in math courses***. Studies show that students are motivated when they see applications. We recommend including minor programming assignments or demonstrations of applications to increase student motivation. While computer science departments may not be able to insert such applications into courses offered by other departments, it is possible to include applications of math in the computer science courses that are co-scheduled with mathematical requirements, and to engage with textbook publishers to provide such material.
- ***Apply available resources to enable student success***. The subcommittee recommends that institutions adopt preparatory options to ensure sufficient background without lowering standards in mathematics. Theory courses can be moved further back in the curriculum to accommodate first-year preparation, for example. And, where possible, institutions can avail of online self-paced tutoring systems alongside regular coursework.
- ***Expand core mathematical requirements to meet the rising demand in new growth areas of computer science***. What is clear, looking forward to the next decade, is that exciting high-growth areas of computer science require a strong background in linear algebra, probability and statistics (preferably calculus-based). Accordingly, we recommend including as much of this material into the standard curriculum as possible,
- ***Send a clear message to students about mathematics while accommodating their individual circumstances***. Faculty and institutions are often under pressure to help every

student succeed, many of whom struggle with math. While pathways, including computer science-adjacent degrees or tracks, can be created to steer students past math requirements towards software-focused careers, faculty should be equally direct in explaining the importance of sufficient mathematical preparation for graduate school and for the very topical areas that excite students.

Networking and Communication (NC)

Text

Operating Systems (OS)

Text

Parallel and Distributed Computing (PDC)

- Should we teach parallel computing in CS I?

Security (SEC)

Text

Society, Ethics and Professionalism (SEP)

- Is introduction to ethical thinking and awareness of social issues sufficient for our graduates to act ethically? If not, does that put the burden on instructors to not only lead discussion about the pressing questions of the day (free speech, filter bubbles, the rise of nationalism, cryptocurrencies, economic disruption of automation) but also to weigh in on those matters? How should that be done? Will we just be imparting our own biases upon our students?
- How could we weave SEP throughout the curriculum in practice? Is this realistic? How much coordination would it take? Is it possible in reality to have experts in ethics, philosophy, etc. *and* CS course X, Y, and Z deliver some of the SEP content for courses X, Y and Z? How much less optimal is it to have a standalone ethics course? Is there another model in between these two extremes (neglecting the super extreme of not having any coordinated or targeted SEP content in our courses)?
- How can we effectively impart the core values of DEIA into our students' education? How is this best done in a CS context? How can we effectively impart the core values and skills of professionalism into our students' education? Are toy projects a suitable context for these? Are work-placements / internships better? Should we put more focus on efforts to having more programs / degrees contain these placements / internships?
- Should software developers be licensed by the state just as engineers, architects, and medical practitioners are? This is an older debate, but given the impact of software systems (akin to safe bridges, buildings, etc), maybe it is time to revisit it. This speaks more to SEP since it addresses the question of assignment of (financial) responsibility to a licensed expert who signed off on a software design or actual code.

- What would a set of current SEP case studies look like?
- Collateral learning of SEP issues?

Software Development Fundamentals (SDF)

- Now that LLMs can automatically generate code for most introductory-level programming problems, do students still need to learn how to write code? Or should they be focusing primarily on reading, critiquing and verifying the correctness of code generated by LLMs? We believe that computer science students still need to be able to write code. How they go about writing code may change rapidly and dramatically with improvements in LLMs.
- Dynamically typed languages for introductory programming?

Software Engineering (SE)

- Are programs in computing that rely on a *single* team project beneficial? Teamwork is ubiquitous in industry, but no teams are formed entirely out of people with similar background and no prior experience working on a team. The heterogeneous background and experience of real times is fundamentally different than the comparative homogeneity of undergrads.
- Do students have sufficient opportunity to practice with open-ended problems, where the choice of tools and approach are relevant (or critical?) In a SE context, most work involves evaluating tradeoffs: between CPU and RAM, between speed-of-implementation and runtime optimization, between DIY and COTS or OSS approaches. Are students given enough opportunity to practice the critical decision making skills necessary to succeed in a professional environment?
- Are our students exposed (in theory or in practice) to the essential role of communication and teamwork skills in professional computing disciplines? If we graduate students that are industry-bound but believe a professional job to be nothing more than programming full-time, we've done them (and their future employers) a huge disservice. We encourage communication exercises and activities, especially in SE environments: participate in code review, and judge on communication skill as well as technical merit, participate in design presentations, etc.
- Software developed in a team setting (software engineering rather than programming) is more likely to have an impact on society for good or ill. At the same time, teamwork means no single person may be responsible for the impact of the software - the larger the project, the less responsibility and the greater the potential impact. How do we instill a proper sense of responsibility for the whole?
- Where do we put the balance on formalism, especially given the limited focus devoted to SE topics? Should we use our limited time to introduce students to ideas that will pay off later in their career, or that are a perfect-but-infrequent solution? Our current approach to validation focuses on attempts to get high-fidelity evidence (unit tests) over proofs and other formal methods, which may be more immediately useful but fails to expose students to interesting long-term ideas.

Specialized Platform Development (SPD)

Text

Systems Fundamentals (SF)

Text

Considerations by Curriculum

- Active learning is an important component of any computer science course – doing helps learning computer science. Courses that use electronic books (ebooks) are a significant improvement over the traditional lecture-based courses that do not involve any active learning component – the provide ample opportunities to apply learning through problem-solving activities. In this vein, it is important to emphasize that ideally, active learning should cover the entire gamut of skill levels – not just *apply*, but also *evaluate* and *develop*.

Curricular Practices

Introduction

Prior curricular reports enumerated issues in the design and delivery of computer science curriculum. Given the increased importance of these issues, the decision was made to provide in the CS2023 curricular report, guidelines for computer science educators to address these issues in their teaching practices. To this end, experts were identified and peer-reviewed, well-researched, in-depth articles were solicited from them to be published under the auspices of CS2023. These articles complement the curricular guidelines in sections 2 and 3: whereas curricular guidelines list what should be covered in the curriculum, these articles describe how and why they should be covered, including challenges, best practices, etc.

The articles may be categorized as covering:

- **Social aspects** such as teaching about accessibility, computer science for social good, and ethics in computer science education;
- **Professional practices** including educational practices in varied settings. In an effort to globalize computer science education, articles were also invited on educational practices in various parts of the world. It is hoped that these articles will foster mutual understanding and exchange of ideas, engender transnational collaboration and student exchange, and serve to integrate computer science education at the global level through shared understanding of the challenges and opportunities of computer science education.
- **Pedagogic considerations** such as CS + X, crosscutting themes in computer science, the impact of large language models (LLMs) on programming instruction and the future of educational materials in computer science.

The articles provide a “lay of the land”, a snapshot of the current state of the art of computer science education as a reference for future scholarship on the teaching and learning of computer science. They are not meant advocate specific approaches or viewpoints, but rather help computer science educators weigh their options and make informed decisions.

The articles have been subjected to peer review when possible. The computer science education community has been invited to provide feedback and suggestions on the first drafts of most of the articles. Many of the articles have been or are in the process of being published in conferences and journals. In this section, abstracts of the articles have been provided. The full articles are accessible at the website csed.acm.org.

Social Aspects

Given the pervasive nature of computing applications, educators would be remiss not to teach their students the principles of responsible computing. How they should go about doing so is explored in the article “**Multiple Approaches for Teaching Responsible Computing**”. It uses research in the social sciences and humanities to nudge responsible computing away from mere post-hoc risk management to an integrated consideration of values throughout the lifecycle of computing products.

In a globalized world, applications of computing transcend national borders. In this context, students in the Global North must develop awareness of transnational complexities whereas those from the Global Souths must adapt to foreign standards and practices. This is brought home by the article "**Making ethics at home in Global CS Education: Provoking stories from the Souths**."

The article "**Computing for Social Good in Education**" highlights how computing education can be used to improve society and address societal needs while also providing authentic computing environments in education and appealing as a discipline to women and groups underrepresented in computing.

Other articles include:

- Teaching about Accessibility in CS education
- Teaching about Diversity, Equity, Inclusion and Justice

Professional Practices

No curricular guidelines are complete by themselves. They must be adapted to local strengths and needs. In this regard, the article on "**Computer science in the liberal arts context**" points a way to adapt CS2023 to the needs of liberal arts colleges that constrain the size of the major in order to allow their students exposure to a broad range of subjects. In the same vein, in Section 2, curricular packaging has been suggested for programs that are 8, 12 and 16 courses large.

Community and technical colleges award academic transfer degrees that enable students to transfer to four-year colleges. They provide an affordable on-ramp to baccalaureate degrees that are attuned to the needs of the local workforce. The article "**Computer Science Education in Community Colleges**" provides a roadmap for how the CS2023 curricular guidelines can be adapted to achieve these objectives.

Other articles include:

- Computer science education practices in geographic areas including the middle east, Africa, Latin America, China and Australasia.

Programmatic Considerations

Several themes such as abstraction, modularity, generalization, and tradeoff cut across the various knowledge areas of computer science. Recognizing and appreciating these themes is essential for developing maturity as computer science professionals. The article "**Connecting Concepts across Knowledge Areas**" enumerates these themes as a helpful guide for educators.

The article “**The Future of Computer Science Educational Materials”** provides a fascinating look into the rapidly changing landscape of educational materials for computer science, including issues of personalization, cloud-based access, integration of Artificial Intelligence, attending to social aspects of learning, catering to underrepresented students, and emphasizing mastery-based learning, to name a few. It provides a peek into the future of computer science education itself.

The article “**The Role of Formal Methods in Computer Science Education**” makes the case for incorporating formal methods in computer science education. It buttresses its case with testimonials from industry.

Other articles include:

- CS + X
- Large Language Models (LLMs) and introductory programming

Multiple Approaches for Teaching Responsible Computing

Stacy A. Doore, Colby College, Colby, ME, USA
Atri Rudra, University at Buffalo, Buffalo, NY, USA
Michelle Trim, University Massachusetts Amherst, Amherst, MA, USA
Joycelyn Streator, Mozilla Foundation, USA
Richard Blumenthal, Regis University, Denver, CO, USA
Bobby Schnabel, University Colorado Boulder, Boulder, CO, USA

Teaching applied ethics in computer science (and computing in general) has shifted from a perspective of teaching about professional codes of conduct and an emphasis on risk management towards a broader understanding of the impacts of computing on humanity and the environment and the principles and practices of responsible computing. This shift has produced a diversity of approaches for integrating responsible computing instruction into core computer science knowledge areas and for an expansion of dedicated courses focused on computing ethics. There is an increased recognition that students need intentional and consistent opportunities throughout their computer science education to develop the critical thinking, analytical reasoning, and cultural competency skills to understand their roles and professional duties in the responsible design, implementation, and management of complex socio-technological systems. Therefore, computing programs are re-evaluating the ways in which students learn to identify and assess the impact of computing on individuals, communities, and societies along with other critical professional skills such as effective communication, workplace conduct, and regulatory responsibilities.

One of the primary shifts in the approach to teaching computing ethics comes from research in the social sciences and humanities. This position is grounded in the idea that all computing artifacts, projects, tools, and products are situated within a set of ideas, attitudes, goals, and cultural norms. This means that all computing endeavors have embedded within them a set of values. Through teaching students critical analysis methods, we can help them to identify potential biases, flaws, and unintentional harms in applications or systems if they can examine the underlying assumptions driving those designs and work with others to correct them. This kind of analysis makes space to bring real world technologies, stakeholders, and domain experts into the classroom for discussion, avoiding the pitfall of only engaging in toy problems. To teach responsible computing always requires us to first recognize that computing happens in a context that is shaped by cultural values, including our own professional culture and values.

The purpose of this paper is to highlight current scholarship, principles, and practices in the teaching of responsible computing in undergraduate computer science settings. The paper is organized around four primary sections: 1) a high-level rationale for the adoption of different pedagogical approaches based on program context and course learning goals, 2) a brief survey of responsible computing pedagogical approaches; 3) illustrative examples of how topics within the CS 2023 Social, Ethical, and Professional (SEP) knowledge area can be implemented and assessed across the broad spectrum of undergraduate computing courses; and 4) links to examples of current best practices, tools, and resources for faculty to build responsible computing teaching into their specific instructional settings and CS2023 knowledge areas.

Making ethics at home in Global CS Education: Provoking stories from the Souths

Marisol Wong-Villacres1, Escuela Superior Politecnica del Litoral, Guayaquil, Ecuador
Cat Kutay, Charles Darwin University, Northern Territory, Australia
Shaimaa Lazem, City of Scientific Research and Technological applications, Alexandria, Egypt
Nova Ahmed, North South University, Dhaka, Bangladesh
Cristina Abad, Escuela Superior Politecnica del Litoral, Guayaquil, Ecuador
Cesar Collazos, Universidad del Cauca, Popayan, Colombia
Shady Elbassuoni, American University of Beirut, Beirut, Lebanon
Farzana Islam, North South University, Dhaka, Bangladesh
Deepa Singh, University of Delhi, New Delhi, India,
Tasmiah Tahsin Mayeesha, North South University, Dhaka, Bangladesh
Martin Mabeifam Ujakpa, Ghana Communication Technology University, Accra, Ghana
Tariq Zaman, University of Technology, Sibu, Malaysia
Nicola J. Bidwell, Charles Darwin University and University of Melbourne, Australia, International University of Management, Namibia

There are few studies about how CS programs should account for the ways ethical dilemmas and approaches to ethics are situated in cultural, philosophical and governance systems, religions and languages (Hughes et al, 2020). We explore some of the complexities that arise for teaching and learning about ethics in the Global Souths, or the geographic and conceptual spaces that are negatively impacted by capitalist globalization and the US-European norms and values exported in computing products, processes and education.

We consulted 46 participants in First Nations Australia, Bangladesh, Brazil, Chile, Colombia, Ecuador, Egypt, Ghana, India, Kenya, Lebanon, Malaysia, Mexico, Namibia and Sri Lanka. Most participants are university educators, but nine are computer industry professionals., We worked in four geographical teams: Africa and the Middle East, Asia-Oceania, Latin America and South-Asia. A group of coordinators (co-authors 1, 2, 3, 4 and 13) decided the main topics to explore, and then each team conducted interviews or administered questionnaires suited to their region.

We organise participants' insights and experience as stories under four main themes. Firstly, ethics relate to diverse perspectives on privacy and institutional approaches to confidentiality. Secondly, people enact ethics by complying with regulations that also attain other goals and difficulties arise in education when regulations are absent or practices are ambiguous. Thirdly, discriminations occur based on people's gender, technical ability and/or minoritised position. Finally, participants' insights suggest a relational rather than transactional approach to ethics.

Ethical guidelines are entangled in socioeconomic circumstances, cultural norms and existing/non-existing policies and structures. Diverse participants explained how their practices cannot align with globalised guidelines and made impressive efforts to fill the gaps and maintain integrity. Thus, guidance should speak to and come "from within" local realities and should focus on leveraging students' values, knowledge and experiences. This requires CS ethics education to promote respect for localised ethical judgements and recognise that approaches in the Global Souths are situated in transnational politics and demands to juggle many factors in managing globalised regulations (Israel, 2017; Png, 2022). To prepare students for careers in a global industry, CS educators in the Global North should ensure students are aware of the many transnational complexities when they introduce examples from the Souths.

CS students in the Global Souths must adapt at an extraordinary pace. Many learn the globalised professional standards, that signal legitimacy, in settings that markedly differ from the places where they were raised, live, or work. At the same time, they must adapt as their nations introduce new rules and regulations (e.g., data protection laws) and their educators negotiate the gaps created by static, anticipatory, globalised ethical codes. Thus, we advocate for In-Action Ethics (Frauenberger et al's, 2017). Rather than focus on ethical principles embedded with a particular ontological stance, In-Action Ethics centres the ways moral positions are embodied in actions. In-Action Ethics can be applied across CS knowledge areas, within and between diverse settings and prompt students to reflect on their actions along their own trajectories in considering what is the right thing to do.

References

Christopher Frauenberger, Marjo Rauhala & Geraldine Fitzpatrick .2017. In-action ethics. Interacting with computers, 29(2), 220-236.

Janet Hughes, Ethan Plaut, Feng Wang, Elizabeth von Briesen, Cheryl Brown, Gerry Cross, Viraj Kumar, & Paul Myers. 2020. Global and local agendas of computing ethics education. In Proceedings of the Conference on Innovation and Technology in Computer Science Education 239-245. ACM.

Mark Israel. 2017. Ethical imperialism? Exporting research ethics to the global south. The Sage handbook of qualitative research ethics, 89-102.

Marie-Therese Png. 2022. At the Tensions of Souths and North: Critical Roles of Global South stakeholders in AI Governance. In Proceedings ACM Conference on Fairness, Accountability, and Transparency 1434-1445.

Computing for Social Good in Education

Heidi J. C. Ellis, Western New England University, Springfield, MA, USA
Gregory W. Hislop, Drexel University, Philadelphia, PA, USA
Mikey Goldweber, Denison University, Granville, OH, USA
Sam Rebelsky, Grinnell College, Grinnell, IA, USA
Janice L. Pearce, Berea College, KY, USA
Patti Ordonez, University of Maryland Baltimore County, MD, USA
Marcelo Pias, Universidade Federal do Rio Grande, Brazil
Neil Gordon, University of Hull, Hull, UK

Computing for Social Good (CSG) encompasses the potential of computing to have a positive impact on individuals, communities, and society, both locally and globally. Computing for Social Good in Education (CSG-Ed) addresses the role of CSG as a component of computing education including the importance of CSG, recommended CSG content, depth of coverage, approaches to teaching, and benefits of CSG in computing education. Also covered are a summary of prior work in CSG-Ed, related topics in computing, and suggestions for implementation of CSG in computing curricula.

The discussion begins with an overview of CSG that expands from defining the term to identifying key topics that are within scope. The focus of computing for social good is the potential of computing to improve society and to address common societal needs such as education, health care, and economic development. This discussion of social good naturally brings to mind the potential for computing to cause harm as well as good. This connection raises a set of closely related topics including professional ethics, and various harms of computing such as algorithmic bias, privacy loss, and ecological impact of computer hardware creation, operation, and disposal.

The importance of CSG in computing education has been recognized for decades, but as with many important topics, CSG is in tight competition for a share of the available curriculum time. On one hand, CSG is recognized as essential and therefore has a place in curricular recommendations, codes of ethics, and accreditation standards. On the other hand, CSG may not be seen as providing knowledge students need immediately to begin a computing career. As such, the amount of time devoted directly to CSG is likely to be rather limited in most computing curricula. This tension informs the discussion of approaches to teaching CSG, where there is consideration of ways to incorporate CSG as part of teaching core computing technical topics.

The discussion also includes a summary of benefits of CSG-Ed including research results in this area. CSG has been used to enhance computing education by providing examples and case studies that embody authentic computing environments. This is especially useful for study of software development and software engineering with regard to both technical skills such as design under constraints and dealing with complexity and also professional skills such as problem solving and communication. CSG has also been shown to impact student motivation and interest. Of particular interest is evidence that incorporating CSG has a strong positive appeal to women and some other underrepresented groups of computing students.

Computing continues to expand world-wide and it touches and is changing more and more aspects of everyday life in simple and profound ways. Graduates of computing degree programs must have an understanding of computing for social good as part of their understanding of computing. This discussion of CSG-Ed summarizes what we know about delivering that understanding in the rapidly changing context of computing education.

Computer Science Curriculum Guidelines: A New Liberal Arts Perspective

Jakob Barnard; University of Jamestown; Jamestown, MD, USA
Valerie Barr; Bard College; Annandale-on-Hudson, NY, USA
Grant Braught; Dickinson College; Carlisle, PA, USA
Janet Davis; Whitman College; Walla Walla, WA, USA
Amanda Holland-Minkley; Washington & Jefferson College; Washington, PA, USA
David Reed; Creighton University; Omaha, NE, USA
Karl Schmitt; Trinity Christian College; Palos Heights, IL, USA
Andrea Tartaro; Furman University; Greenville, SC, USA
James Teresco; Siena College; Loudonville, NY, USA

ACM/IEEE curriculum guidelines for computer science, such as CS2013 or the forthcoming CS2023, provide well-researched and detailed guidance regarding the content and skills that make up an undergraduate computer science (CS) program. Liberal arts CS programs often struggle to apply these guidelines within their institutional and departmental contexts. Historically, this has been addressed through the development of model CS curricula tailored for the liberal arts context. We take a different position: that no single model curriculum can apply across the wide range of liberal arts institutions. Instead, we argue that liberal arts CS educators need best practices for using guidelines such as CS2023 to inform curriculum design. These practices must acknowledge the opportunities and priorities of a liberal arts philosophy as well as institutional and program missions, priorities, and identities.

The history, context, and data about liberal arts CS curriculum design support the position that the liberal arts computing community is best supported by a process for working with curricular guidelines rather than a curriculum model or set of exemplars. Previous work with ACM/IEEE curriculum guidelines over the decades has trended towards acknowledging the variety of forms liberal arts CS curricula may take and away from presenting a unified “liberal arts” model. A review of liberal arts CS programs demonstrates how institutional context, including institutional mission and structural factors, shape their curricula. Survey data indicates that liberal arts programs have distinct identities or missions, and this directly impacts curriculum and course design decisions. Programs prioritize flexible pathways through their programs coupled with careful limits on required courses and lengths of prerequisite chains. This can drive innovative course design where content from Knowledge Areas is blended rather than compartmentalized into distinct courses. The CS curriculum is viewed as part of the larger institutional curriculum and the audience for CS courses is broader than just students in the major, at both the introductory level and beyond.

To support the unique needs of CS liberal arts programs, we propose a process that guides programs to work with CS2023 through the lens of institutional and program missions and identities, goals, priorities and situational factors. The Process Workbook we have developed comprises six major steps:

1. articulate institutional and program mission and identity;
2. develop curricular design principles driven by identity and structural factors, with attention to diversity, equity, and inclusion;

3. identify aspirational learning outcomes in response to principles and identity;
4. determine the alignment of the current curriculum with CS2023;
5. evaluate the current program, with attention to current strengths, unmet goals, and opportunities for improvement;
6. design, implement, and assess changes to the curriculum.

An initial version of the Process Workbook, based on our research and feedback from workshops and pilot usage within individual departments, is available as a supplement to this article. The authors will continue this iterative design process and release additional updates as we gather more feedback. Future work includes development of a repository of examples of how programs have made use of the Workbook to review and redesign their curricula in the light of CS2023.

Computer Science Education in Community Colleges

Elizabeth Hawthorn, Rider University, Lawrenceville, NJ, USA
Lori Postner, Nassau Community College, Garden City, NY, USA
Christian Servin, El Paso Community College, El Paso, TX, USA
Cara Tang, Portland Community College, Portland, OR, USA
Cindy Tucker, Bluegrass Community and Technical College, Lexington, KY, USA

Community and Technical Colleges serve as two-year educational institutions, providing diverse academic degrees like associate's degrees in academic and applied sciences, certificates of completion, and remedial degrees. These colleges play a crucial role in fostering collaboration between students, workers, and institutions through educational and workforce initiatives. Over the past 50+ years, Community Colleges have served as a hub for various educational initiatives and partnerships involving K-12 schools, four-year colleges, and workforce/industry collaborations.

These colleges offer specialized programs that help students focus on specific educational pathways. Among the programs available, computing-related courses are prominent, including Computer Science degrees, particularly the Associate in Arts (AA) and Sciences (AS) degrees, known as academic transfer degrees. These transfer degrees are designed to align with the ACM/IEEE curricular guidelines, primarily focusing on creating two-year programs that facilitate smooth transferability to four-year colleges.

Furthermore, the computing programs offered by Community Colleges are influenced by the specific needs and aspirations of the regional workforce and industry. Advisory boards and committees play a significant role in shaping these programs by providing recommendations based on the demands of the job market. While the ACM Committee for Computing in Community Colleges (CCECC) and similar entities help address inquiries related to these transfer degrees, there is a desire to capture the challenges, requirements, and recommendations from the Community College perspective in developing general curricular guidelines.

This work presents the context and perspective of a community college education during the design and development of curricular guidelines, exemplified by the ACM/IEEE/AAAI CS2023 project. It emphasizes the importance of understanding the unique challenges faced by Community Colleges and their specific needs while formulating curricular guidelines. Additionally, the paper envisions considerations for future years regarding curricular development and administrative efforts, considering the evolving educational landscape and industry demands. By doing so, the vision is to enhance the effectiveness and relevance of computing programs offered by Community Colleges and foster better alignment with the needs of students and the job market.

[1] Christian Servin, Elizabeth K. Hawthorne, Lori Postner, Cara Tang, and Cindy Tucker.

2023. Community Colleges Perspectives: From Challenges to Considerations in Curricula Development. In Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 2 (SIGCSE 2023). Association for Computing Machinery, New York, NY, USA, 1244. https://doi.org/10.1145/3545947.3573335

[2] Elizabeth Hawthorne, Cara Tang, Cindy Tucker, and Christian Servin. 2017. Computer Science Curricular Guidelines for Associate-Degree Transfer Programs (Abstract Only). In Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education (SIGCSE '17). Association for Computing Machinery, New York, NY, USA, 725. https://doi.org/10.1145/3017680.3022348

Connecting Concepts across Knowledge Areas

Ramachandran Vaidyanathan, Louisiana State University, Baton Rouge, LA
Jerry L. Trahan, Louisiana State University, Baton Rouge, LA
Suresh Rai, Louisiana State University, Baton Rouge, LA
Sheikh Ghafoor, Tennessee Tech. University, Cookeville, TN
Alan Sussman, University of Maryland, College Park, MD
Charles Weems, University of Massachusetts, Amherst, MD
Amruth Kumar, Ramapo College, Mahwah, NJ

We identify fundamental concepts that we believe students in computing should be familiar with. These concepts cut across various computational topics and can be introduced to students in a variety of settings. This work seeks to map these concepts to a set of courses that roughly align with the ACM/IEEE CS2023 Knowledge-Areas. It also provides suggestions to teach these concepts across course sequences.

As an example, the concept of a machine state can be conveyed in many ways across many courses. In digital logic a state may be flip-flop outputs, whereas in programming a set of variables may indicate state. Similar ideas can be explored in algorithms, systems and the theory of computation. Here the key idea conveyed to students is that the state is that portion of the past that the machine needs to take the next step.

On the whole, a goal of this work is to facilitate the placement of “concept-dots" across the curriculum, with “dots" placed in upstream courses being “connected" in advanced downstream courses to make key concepts clear. We expect such an effort to provide students a deeper understanding of the concepts, and broader perspective on their applicability.

The Future of Computer Science Educational Materials

Peter Brusilovsky, University of Pittsburgh, PA, USA
Barbara Ericson, University of Michigan, USA
Cay Horstmann, PFH Göttingen, Germany
Craig Johnson, University of Illinois at Urbana-Champaign, IL, USA
Christian Servin, El Paso Community College, TX, USA
Frank Vahid, University of California Riverside, CA, USA

CS education relies on diverse educational materials like textbooks, presentation slides, labs, and test banks, which have evolved significantly over the past two decades. New additions, such as videos, animations, online homework systems, and auto-graded programming exercises, aim to enhance student success and elevate the instructor's role. This article explores the future of educational materials in CS education, focusing on effective approaches. A prominent trend is the growing interactivity of educational materials, providing immediate feedback to students throughout their learning journey. Integrating artificial intelligence allows these materials to adapt to individual learners and offer valuable assistance. Many educational resources are transitioning to cloud-based platforms, enabling continuous data collection and analysis for improvement.

Another essential aspect is the emphasis on supporting the social aspects of learning, promoting peer collaboration and coaching. Open education resources (OER) and products from educational technology companies are expanding, leading to a demand for customization, content creation, and seamless sharing of high-quality materials. Automation is increasingly streamlining class administration, and the importance of tool interoperability is growing. Learning management systems (LMS) are continually improving to accommodate the changing landscape of educational materials. Moreover, there's a rising focus on developing materials that cater to traditionally underrepresented students, fostering inclusivity and diversity in CS education.

Educational materials are shifting towards supporting mastery-based learning, prioritizing student success over performance-based filtering. However, the inclusion of both sanctioned and unsanctioned online resources, like homework help websites and video sharing sites, poses challenges and opportunities for instructors. This work aims to outline the future of educational materials in CS education, encouraging educators and administrators to recognize areas for development, adaptation, and innovation. Embracing these changes will be crucial in ensuring a more effective and inclusive CS education experience as the landscape continues to evolve.

[1] Peter Brusilovsky, Barbara J. Ericson, Cay S. Horstmann, Christian Servin, Frank Vahid, and Craig Zilles. 2023. Significant Trends in CS Educational Material: Current and Future. In Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 2 (SIGCSE 2023). Association for Computing Machinery, New York, NY, USA, 1253. https://doi.org/10.1145/3545947.3573353

The Role of Formal Methods in Computer Science Education

Maurice ter Beek, Manfred Broy, Brijesh Dongol, Emil Sekerinski, et al

Formal Methods provide a wide range of techniques and tools for specifying, developing, analysing, and verifying software and hardware systems. In the papers, we make four key points: (1) Every Computer Science graduate needs to have an education in Formal Methods; (2) Formal Methods can support Teamwork, Code Review, Software Testing, and more; (3) Formal Methods are applicable in numerous domains (not only in safety-critical applications); and (4) The current offering of Formal Methods in the Computer Science education is inadequate.

Computer Science, namely the science of solving problems with software and software-intensive systems, provides the knowledge and skills to understand and capture precisely what a situation requires, and then develop a formal solution in a programming language. The most fundamental skill of a computer scientist, that of *abstraction*, is best addressed by Formal Methods. They provide the rigor for reasoning about goals, such as validation and verification, thus guaranteeing adequacy, accuracy, and correctness of implementations.

Formal Methods thinking, i.e., the ideas from Formal Methods applied in informal, lightweight, practical, and accessible ways, should be part of the recommended curriculum for every Computer Science student. Even students who train only in that “thinking” will become much better programmers. In addition, there are students who, exposed to those ideas, will be ideally positioned to study more: why the techniques work; how they can be automated; and how new ones can be developed. They could follow subsequently an optional path, including topics such as semantics, logics, and proof-automation techniques.

Formal Methods were conceived for teaching programming to novices more effectively than by informal reasoning and testing. Formal Methods explain algorithmic problem solving, design patterns, model-driven engineering, software architecture, software product lines, requirements engineering, and security. Formalisms can concisely and precisely express underlying fundamental design principles and equip programmers with a tool to handle related problems.

Formal Methods are becoming widely applied in industry, from eliciting requirements and early design to deployment, configuration, and runtime monitoring. Successfully applying Formal Methods in industry ranges from well-known stories in the safety-critical domain, such as railways and other transportation domains, to areas such as lithography manufacturing and cloud security in e-commerce, for example. Testimonies come from representatives who, either directly or indirectly, use or have used Formal Methods in their industrial project endeavours. Importantly, they are spread geographically, including Europe, Asia, North and South America.

ACM-CS2023 is the ideal time and place to adjust the way we teach Computer Science. There are mature tools and proofs of concept available and the possibility of designing coherent teaching paths. Importantly, this can be done without displacing the other “engineering” aspects of Computer Science already widely accepted as essential. Support for teachers is available.

Acknowledgments

General

The following assisted the task force in its work:

- ACM Staff:
  - Yan Timanovsky, ACM Education & Professional Development Manager
  - Lisa Kline, ACM Education and Professional Development Assistant
  - John Otero, Site Selection
- Jens Palsberg, University of California, Los Angeles, CA, USA, Chair of SIG Chairs
- Mehran Sahami, Stanford University, Stanford, CA, USA, ACM Co-Chair, CS 2013
- ACM Education Board
  - Elizabeth Hawthorne, Rider University, Lawrenceville, NJ, USA
  - Alison Derbenwick Miller, Oracle Inc.
  - Christine Stephenson, Google Inc. (retired)
- Mihaela Sabin, University of New Hampshire, Manchester, NH, USA
- Bruce McMillin, IEEE Computer Society, Professional & Educational Activities Board – Curriculum and Accreditation Committee (CA), Chair
- IEEE Computer Society Staff:
  - Eric Berkowitz, Director of Membership, USA
  - Michelle Phon, Certification and Professional Education, USA

Reviewers

The following reviewed various drafts of the curricula:

- Ginger Alford, Southern Methodist University, Dallas, TX, USA
- Jeannie Albrecht, Williams College, Williamstown, MA, USA
- Mostafa Ammar, Georgia Institute of Technology, Atlanta, GA, USA
- Tom Anderson, University of Washington, Seattle, WA, USA
- Christopher Andrews, Middlebury College, Middlebury, VT, USA
- Elisa Baniassad, The University of British Columbia, Vancouver, BC, Canada
- Arvind Bansal, Kent State University, Kent, OH, USA
- Phillip Barry, University of Minnesota, Minneapolis, MN, USA
- Brett A. Becker, University College Dublin, Dublin, Ireland
- Judith Bishop, Stellenbosch University, Stellenbosch, South Africa
- Alan Blackwell, University of Cambridge, Cambridge, UK
- Olivier Bonaventure, Université Catholique de Louvain, Louvain-la-Neuve, Belgium
- Kim Bruce, Pomona College, Claremont, CA, USA
- John Carroll, Penn State, University Park, PA, USA
- Gennadiy Civil, Google Inc., New York, NY, USA

- Thomas Clemen, Hamburg University of Applied Sciences, Hamburg, Germany
- Jon Crowcroft, University of Cambridge, Cambridge, UK
- Melissa Dark, Dark Enterprises, Inc., Lafayette, IN, USA
- Arindam Das, Eastern Washington University, Cheney, WA, USA
- Karen C. Davis, Miami University, Oxford, OH, USA
- Henry Duwe, Iowa State University, Ames, IA, USA
- Roger D. Eastman, University of Maryland, College Park, MD, USA
- Yasmine Elglaly, Western Washington University, Bellingham WA, USA
- Trilce Estrada, University of New Mexico, Albuquerque, NM, USA
- David Flanagan, Text book Author
- Akshay Gadre, University of Washington, Seattle, WA, USA
- Ed Gehringer, North Carolina State University, Raleigh, NC, USA
- Sheikh Ghafoor, Tennessee Tech University, Cookville, TN, USA
- Tirthankar Ghosh, University of New Haven, West Haven, CT, USA
- Michael Goldwasser, Saint Louis University, St. Louis, MO, USA
- Martin Goodfellow, University of Strathclyde, Glasgow, UK
- Vikram Goyal, IIIT, Delhi, India
- Xinfei Guo, Shanghai Jiao Tong University, Shanghai, China
- Anshul Gupta, IBM Research, Yorktown Heights, NY, USA
- Sally Hamouda, Virginia Tech, Blacksburg, VA, USA
- Matthew Hertz, University at Buffalo, Buffalo, NY, USA
- Michael Hilton, Carnegie Mellon University, Pittsburgh, PA, USA
- Bijendra Nath Jain, IIIT, Delhi, India
- Kenneth Johnson, Auckland University of Technology, Auckland, New Zealand
- Krishna Kant, Temple University, Philadelphia, PA, USA
- Hakan Kantas, Halkbank, Istanbul, Turkiye
- Amey Karkare, Indian Institute of Technology, Kanpur, India
- Kamalakar Karlapalem, International Institute of Information Technology, Hyderabad, India
- Theodore Kim, Yale University, New Haven, CT, USA
- Michael S. Kirkpatrick, James Madison University, Harrisonburg, VA, USA
- Tobias Kohn, Vienna University of Technology, Vienna, Austria
- Eleandro Maschio Krynski, Universidade Tecnológica Federal do Paraná, Guarapuava, Paraná, Brazil
- Ludek Kucera, Charles University, Prague, Czechia
- Fernando Kuipers, Delft University of Technology, Delft, The Netherlands
- Matthew Fowles Kulukundis, Google, Inc., New York, NY, USA
- Zachary Kurmas, Grand Valley State University, Allendale, MI, USA
- Rosa Lanzilotti, Università di Bari, Bari, Italy
- Alexey Lastovetsky, University College Dublin, Dublin, Ireland
- Gary T. Leavens, University of Central Florida, Orlando, FL, USA
- Kent D. Lee, Luther College, Decorah, IA, USA
- Bonnie Mackellar, St. John’s University, Queens, NY, USA
- Alessio Malizia, Università di Pisa, Pisa, Italy
- Sathiamoorthy Manoharan, University of Auckland, Auckland, New Zealand

- Maristella Matera, Politecnico di Milano, Milano, Italy
- Stephanos Matsumoto, Olin College of Engineering, Needham, MA, USA
- Paul McKenney, Facebook, Inc.
- Michael A. Murphy, Coastal Carolina University, Conway, SC, USA
- Raghava Mutharaju, IIIT, Delhi, India
- V. Lakshmi Narasimhan, Georgia Southern University, Statesboro, GA, USA
- Peter Pacheco, University of San Francisco, San Francisco, CA, USA
- Andrew Petersen, University of Toronto, Mississauga, Canada
- Cynthia A Phillips, Sandia National Lab, Albuquerque, NM, USA
- Benjamin C. Pierce, University of Pennsylvania, Philadelphia, PA, USA
- Sushil K. Prasad, University of Texas, San Antonio, TX, USA
- Rafael Prikladnicki, Pontificia Universidade Catolica do Rio Grande do Sul, Porto Alegre, Brazil
- Keith Quille, Technological University Dublin, Dublin, Ireland
- Catherine Ricardo, Iona University, New Rochelle, NY, USA
- Luigi De Russis, Politecnico di Torino, Torino, Italy
- Beatriz Sousa Santos, University of Aveiro, Aveiro, Portugal
- Michael Shindler, University of California, Irvine, CA, USA
- Ben Shneiderman, University of Maryland, College Park, MD, USA
- Anna Spagnolli, Università di Padova, Padova, Italy
- Davide Spano, Università di Cagliari, Cagliari, Italy
- Anthony Steed, University College London, London, UK
- Michael Stein, Metro State University, Saint Paul, MN, USA
- Alan Sussman, University of Maryland, College Park, MD, USA
- Andrea Tartaro, Furman University, Greenville, SC, USA
- Tim Teitelbaum, Cornell University, Ithaca, NY, USA
- Joseph Temple, Coastal Carolina University, Conway, SC, USA
- Ramachandran Vaidyanathan, Louisiana State University, Baton Rouge, LA, USA
- Salim Virji, Google Inc., New York, NY, USA
- Guiliana Vitiello, Università di Salerno, Salerno, Italy
- Philip Wadler, The University of Edinburgh, Edinburgh, UK
- Charles Weems, University of Massachusetts, Amherst, MA, USA
- Xiaofeng Wang, Free University of Bozen-Bolzano, Bolzano, Italy
- Miguel Young de la Sota, Google Inc., USA
- Massimo Zancanaro, Università di Trento, Trento, Italy
- Ming Zhang, Peking University, Beijing, China

References

1. Atchison, W. F., Conte, S. D., Hamblen, J. W., Hull, T. E., Keenan, T. A., Kehl, W. B., McCluskey, E. J., Navarro, S. O., Rheinboldt, W. C., Schweppe, E. J., Viavant, W., and Young, D. “Curriculum 68: Recommendations for academic programs in computer science.” *Communications of the ACM*, 11, 3 (1968): 151–197.
2. Austing, R. H., Barnes, B. H., Bonnette, D. T., Engel, G. L., and Stokes, G. “Curriculum ’78: Recommendations for the undergraduate program in computer science.” *Communications of the ACM*, 22, 3 (1979): 147–166.
3. ACM/IEEE-CS Joint Curriculum Task Force. “Computing Curricula 1991.” (New York, USA: ACM Press and IEEE Computer Society Press, 1991).
4. ACM/IEEE-CS Joint Curriculum Task Force. “Computing Curricula 2001 Computer Science.” (New York, USA: ACM Press and IEEE Computer Society Press, 2001).
5. ACM/IEEE-CS Interim Review Task Force. “Computer Science Curriculum 2008: An interim revision of CS 2001.” (New York, USA: ACM Press and IEEE Computer Society Press, 2008).
6. ACM/IEEE-CS Joint Task Force on Computing Curricula. “Computing Science Curricula 2013.” (New York, USA: ACM Press and IEEE Computer Society Press, 2013).
7. Sabin, M., Alrumaih, H., Impagliazzo, J., Lunt, B., Zhang, M., Byers, B., Newhouse, W., Paterson, W., Tang, C., van der Veer, G. and Viola, B. Information Technology Curricula 2017: Curriculum Guidelines for Baccalaureate Degree Programs in Information Technology. Association for Computing Machinery, New York, NY, USA, (2017).
8. Clear, A., Parrish, A., Impagliazzo, J., Wang, P., Ciancarini, P., Cuadros-Vargas, E., Frezza, S., Gal-Ezer, J., Pears, A., Takada, S., Topi, H., van der Veer, G., Vichare, A., Waguespack, L. and Zhang, M. Computing Curricula 2020 (CC2020): Paradigms for Future Computing Curricula. Technical Report. Association for Computing Machinery / IEEE Computer Society, New York, NY, USA, (2020).
9. Leidig, P. and Salmela, H. A Competency Model for Undergraduate Programs in Information Systems (IS2020). Technical Report. Association for Computing Machinery, New York, NY, USA, (2021).
10. Danyluk, A. and Leidig, P. Computing Competencies for Undergraduate Data Science Curricula (DS2021). Technical Report. Association for Computing Machinery, New York, NY, USA, (2021).
11. https://iiitd.ac.in/sites/default/files/docs/aicte/AICTE-CSE-Curriculum-Recommendations-July2022.pdf, last accessed July 2023.
12. Prasad, S. K., Estrada, T., Ghafoor, S., Gupta, A., Kant, K., Stunkel, C., Sussman, A., Vaidyanathan, R., Weems, C., Agrawal, K., Barnas, M., Brown, D. W., Bryant, R., Bunde, D. P., Busch, C., Deb, D., Freudenthal, E., Jaja, J., Parashar, M., Phillips, C., Robey, B., Rosenberg, A., Saule, E., Shen, C. 2020. NSF/IEEE-TCPP Curriculum Initiative on Parallel and Distributed Computing - Core Topics for Undergraduates, Version II-beta, Online: http://tcpp.cs.gsu.edu/curriculum/, 53 pages.
13. https://ccecc.acm.org/files/publications/Cyber2yr2020.pdf, last accessed July 2023.
14. https://www.computer.org/volunteering/boards-and-committees/professional-educational-activities/software-engineering-competency-model, last accessed July 2023.
15. Amruth N. Kumar, Brett A. Becker, Marcelo Pias, Michael Oudshoorn, Pankaj Jalote, Christian Servin, Sherif G. Aly, Richard L. Blumenthal, Susan L. Epstein, and Monica D. Anderson. 2023. A

Combined Knowledge and Competency (CKC) Model for Computer Science Curricula. ACM Inroads 14, 3 (September 2023), 22–29. https://doi.org/10.1145/3605215

16. Clear, A., Clear, T., Vichare, A., Charles, T., Frezza, S., Gutica, M., Lunt, B., Maiorana, F., Pears, A., Pitt, F., Riedesel, C. and Szynkiewicz, J. Designing Computer Science Competency Statements: A Process and Curriculum Model for the 21st Century. In Proceedings of the Working Group Reports on Innovation and Technology in Computer Science Education (ITiCSE-WGR '20). Association for Computing Machinery, New York, NY, USA, (2020), 211–246.
17. Frezza, S., Daniels, M., Pears, A., Cajander, A., Kann, V., Kapoor, A., McDermott, R., Peters, A., Sabin, M. and Wallace, C. Modelling Competencies for Computing Education beyond 2020: A Research Based Approach to Defining Competencies in the Computing Disciplines. In Proceedings Companion of the 23rd Annual ACM Conference on Innovation and Technology in Computer Science Education (Larnaca, Cyprus) (ITiCSE 2018 Companion). Association for Computing Machinery, New York, NY, USA, (2018), 148–174.
18. Anderson, Lorin W. and Krathwohl, David R., eds. (2001). A taxonomy for learning, teaching, and assessing: A revision of Bloom's taxonomy of educational objectives. New York: Longman. ISBN 978-0-8013-1903-7.
19. Adeleye Bamkole, Markus Geissler, Koudjo Koumadi, Christian Servin, Cara Tang, and Cindy S. Tucker. "Bloom’s for Computing: Enhancing Bloom's Revised Taxonomy with Verbs for Computing Disciplines". The Association for Computing Machinery. (January 2023). https://ccecc.acm.org/files/publications/Blooms-for-Computing-20230119.pdf