Software Development Fundamentals (SDF)

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

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

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

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

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

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

SDF/Algorithms and Design

[11 Core-Tier1 hours]

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

Topics:

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

Learning Outcomes:

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

SDF/Fundamental Programming Concepts

[10 Core-Tier1 hours]

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

Topics:

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

- Functions and parameter passing
- The concept of recursion

Learning Outcomes:

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

SDF/Fundamental Data Structures

[12 Core-Tier1 hours]

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

Topics:

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

Learning Outcomes:

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

SDF/Development Methods

[9 Core-Tier1 hours]

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

Topics:

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

Learning Outcomes:

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