CS 2013

Software Development Fundamentals (SDF)

Fluency in the process of software development is a prerequisite to the study of most of computer science. Beyond programming skills, undergraduate programs must also teach students to design and analyze algorithms, select appropriate paradigms, and utilize 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.

The organization of this knowledge area should not be misinterpreted as prescribing a particular approach to the first year programming sequence. A variety of courses could be developed that would cover this material in the first year, integrating topics from related areas to fill out the sequence. For example, it is expected that an instructor would select one or more programming paradigms (e.g., object-oriented programming in Java, functional programming in Scheme, scripting in Python) to illustrate these software development concepts and skills. Likewise, an instructor could choose to emphasize more 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.

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

	Core-Tier1 hours	Core-Tier2 hours	Includes Electives
SDF/FundamentalConcepts	**11**		**N**
SDF/FundamentalDataStructures	**10**		**N**
SDF/AlgorithmsAndDesign	**12**		**N**
SDF/DevelopmentMethods	**9**		**N**

SDF/FundamentalConcepts [11 Core-Tier1 hours]

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
- Structured decomposition
- 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. Apply the techniques of structured (functional) decomposition to break a program into smaller pieces. [Application]
8. Describe the concept of recursion and give examples of its use. [Knowledge]
9. Identify the base case and the general case of a recursively defined problem. [Evaluation]

SDF/FundamentalDataStructures [10 Core-Tier1 hours]

Topics:

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

Learning Objectives:

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/AlgorithmsAndDesign [12 Core-Tier1 hours]

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
    - Encapsulation and information hiding

- Separation of behavior and implementation

Learning Objectives:

1. Discuss the importance of algorithms in the problem-solving process. [Knowledge]
2. Recognize that 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. Identify the data components and behaviors of multiple abstract data types. [Application]
9. Implement a coherent abstract data type, with loose coupling between components and behaviors. [Application]
10. Identify the relative strengths and weaknesses among multiple designs or implementations for a problem. [Evaluation]

SDF/DevelopmentMethods [9 Core-Tier1 hours]

Topics:

- Program correctness
  - 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
  - Unit testing
- Modern programming environments
  - Programming using library components and their APIs
- Debugging strategies
- Documentation and program style

Learning Objectives:

1. Explain why the creation of correct program components is important in the production of quality software. [Knowledge]
2. Be aware of 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

maintainabilty of software. [Application]