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