Writing in the Computer Science Major

Table of Contents

This table summarizes the use of the different writing tasks across the courses in the Computer Science program.

	1st Year			Mid- and Upper Level									Senior	
	CS1 115	CS2 116	Found. 111	Alg. 215	Discrete Math	Org. 220	DBs 32 5	P.L. 335	O.S. 345	A.I. 255	Graph. 365	Theory 375	Capstone	Thesis 500
Summaries	**X**	**X**		**X**		**X**	**X**	**X**	**X**	**X**	**X**		**X**	
PRE/POST	**X**	**X**		**X**		**X**	**X**	**X**	**X**	**X**	**X**		**X**	
Invariants	**X**	**X**	**X**	**X**										
README		**X**		**X**		**X**	**X**	**X**	**X**	**X**	**X**		**X**	
User's Guide							**(X)**				**X**		**X**	
Functional Specs													**X**	
Detailed Specs													**X**	
Proofs			**X**		**X**					**X***		**X**		
Research Paper	**X***					**X**	**(X)**			**(X)**				
Thesis														**X**

X Offered in this course **X*** Offered by some professors in this course **(X)** Under consideration

Introduction

Definitions

Like many disciplines, Computer Science relies on a specific set of definitions when describing its activities. We include some of these keywords and the discipline-specific definitions that appear in this document.

debug – to find and fix mistakes in a computer program
program – also called software or code; the set of terse instructions given to a computer and written in a particular language such as C++, Java, Flash, or Perl.
programmer – also called software engineer; those individuals who write computer programs

Statement of Purpose

Writing in Computer Science is a central part of the discipline in several ways, each wrapped around the concept of the *communication* between programmers, users, researchers, and managers. Programmers need to create written documentation within the body of a program in order to debug it, maintain it, and eventually allow for its efficient transfer into the hands of others. Programmers also need to be able to communicate about the work to a broader audience; managers and professionals need to understand the program in a quick glance, and novice software users need to know how to make use of the program without understanding the internals of how it works. Furthermore, advanced Computer Scientists must enlarge their understanding of the field and how others are continually enriching it by doing research and processing that research through the composition of papers that synthesize what was read.

Because of the inherent rigor and levels of abstraction in our discipline, we value the ability to communicate in our field. This document details the many writing tasks mastered by a Computer Science major in our program, and outlines the ways in which we strive to continue the expansion of our program in this area.

Revision History

May-June, 2005 – Lisa Michaud
The original documents authored by Gousie, LeBlanc, and Michaud were compiled into this format, edited, and extended.

November 2007 – Mark LeBlanc, Mike Gousie
Document revisited. Faculty confirmed the types of writing and writing instruction currently in practice and where each type appears within the curriculum.

Writing Tasks in Computer Science

This section discusses in detail the many writing tasks which are central to the Computer Science curriculum. It includes both the format of the writing task and the audience for which it is aimed, an attribute which in each case places unique demands upon the writer.

Documentation

The act of programming entails the communication of instructions to a computer; however, the artificial and often obfuscated structures of artificial programming languages do not always communicate design, intent, or functionality to the human reader or user. This is the purpose of documentation. We address the creation of documentation in many forms throughout the spectrum of courses in the major. At all levels, we emphasize that documentation must be correct, precise, and complete, avoiding errors in representation, ambiguity, and omissions.

Planning to Program: Functional and Detailed Specifications

The beginning of a large programming project in the "real world" starts with the Functional Specification. This is a document which contains a precise and yet high-level description of the functionality of the overall system and each of its component modules. The audience of this document includes supervisors, marketing departments, and clients or end-users of the system, so this description is crafted in a way that thoroughly describes each action of the future system, and yet is left *purposefully* free from precise technical detail. This is a planning document, and it is debated extensively during pre-implementation meetings. In the practice of crafting such a document, the writer(s) must work through a process including a large number of drafts until all parties are satisfied.

Once the project moves into the implementation phase, the Detailed Specification is designed with the programmers themselves as the intended audience. Now the technical detail is the primary focus, as the specification marks out exactly how each piece of the system should be implemented, down to explanations of each argument in each function prototype. This document references the Functional Specification as is appropriate. One challenge to maintaining this document is that all details cannot always be precisely foreseen; therefore, as the implementation changes from the plan, the document must keep in step, as it will continue to be used as a reference for all others also working on the system.

Within the Program: Summaries, Preconditions, Postconditions, and Invariants

Programming languages allow for "comments," which are essentially asides to the human reader that are ignored by the computer. Used in a systematic way to annotate the structure of the program, these comments provide an internal documentation within the program file which informs the reader of the intent and purpose of the programmer. We emphasize the following forms of internal documentation:

- *Program Summaries.* Intended for the recipient of the program file who may or may not peruse its contents (such as managers or other programming professionals), a short paragraph statement at the top of the program serves as an abstract for the entire document, explaining the purpose of the program and its basic nature. This should be concise, and yet communicate sufficient information for the reader to understand the purpose of the file. A good summary also lists the program's use and creation of other files while executing. This includes a section on INPUT (the data files to be used by this program), OUTPUT (the files to be created to hold the final results), Date Last Modified, and Current Status. An template-example of a program summary is shown below.

```
/*----------------------------------------------------------
Programmer: Ima Hacker

Summary: A program to find all possible motifs of lengths
          [4-7] in a .fna (fasta format) file of nucleotides (DNA)…

INPUT:  (two sources)
       stdin (keyboard): describe expected input and what
                          happens on bad input
       .fna (DNA file):  describe 70 chars per line then
```

```
                newline until EOF; Fasta format!

OUTPUT:   (two places)
        stdout (screen):
        File: mention output filename convention
              (often good to show an example output here)

Date Last Modified:
  07/30/2007(mdl) - started work ….
  08/02/2007(mdl) - trouble with Encrypt function …

STATUS:  as of 08/05/2007 all functions work
         except Decrypt(); I think it is because …
-------------------------------------------------------*/
```

- *Preconditions and Postconditions.* Programming in a modular fashion with subprograms or functions is a key aspect of program design. As programmers design these modules, the intent is to create reusable components, both for the programmer herself and potentially for other programmers as well. Each such module must therefore be accompanied by a *precondition* which documents that which the module assumes to be true before it is executed (and, presumably, is essential to its correct use), as well as a *postcondition* which explains that which the module achieves. Again, the objective is to be succinct, and yet to effectively communicate sufficient information *for another programmer* to be able to use the module without knowing the exact details of the software in the module. (These also aid the original programmer when she is attempting to debug her own program, as violated preconditions are a very likely cause of error.)

```
   /-----\
--/ Swap  \---------------------------------------------------
|
| Summary: Swap the values pointed to by two argument pointers
|
| PRE:  Assumes two pointers point to valid memory addresses
|               and each location holds a long value.
|
| POST: Values pointed to be each pointer are swapped.
|
| SIDE EFFECT: none
-------------------------------------------------------------*/
```

- *Invariants.* Within the module, certain sections of the program can be further documented by invariant statements which describe what should be true throughout that block of program code – for example, the idea that a certain variable's value will not exceed 10, or that another variable will hold the total of all values encountered so far. These statements serve multiple purposes, including aiding in the programmer's ability to identify the source of errors if it can be seen that some aspect of the program violates the invariant statement.

```
i   =  1 ;
sum =  0 ;

// INV: sum == 0 + 1 + 2 + ... + (i-1)  &&  1 <= i <= N+1
while(i <= N) {
       sum += i;
       i++;
}
```

Invariants also serve as the initial step in a rigorous proof that a section of a program is correct.

Accompanying the Program: The README and the User's Guide

The intended audience of the "README" that usually accompanies a program as a separate file (bearing a name intended to draw the eye first) varies according to the nature of the program; essentially, though, it often includes a repetition of the material from the Program Summary as a preface to the essential instructions that must be followed in order to install and set up the program to run.

The User Guide is the next step of abstraction; intended to facilitate the use of a complete software project by a novice, this polished and extended document steps the user through every detail required to perform any function of the program. It must be clear, complete, and coherent.

Mathematical Proofs

As in the field of Mathematics from which it was born, Computer Science often exercises the use of a proof as an essential tool for illustrating a point, exploring the nature of a concept, or verifying the truth of a statement for a range of values. The writing in a proof must be very precise and must correctly use the notation that is conventional for its topic; more importantly, however, it must construct an argument, building points in concrete, defensible steps, until the objective statement is obtained and supported by the work.

Writing correct and succinct proofs is a difficult task and mastery requires repeated exposure. For that reason, instruction with writing proofs appears in multiple courses in the curriculum.

Research Papers

Several of the 200- and 300-level courses have implemented research paper assignments. These expose students to the inherently difficult task of reading and decoding current scientific journal articles, in order to synthesize the ideas and present them in turn. Students must take the time to not just "copy" the material but shape it into a coherent whole in order to communicate to another the essential points, how they might be tied together, and how the central ideas relate to other work.

In these assignments, as with the examples above, we stress the importance of the intended *audience* as an inseparable aspect of the writing process. Specific guidelines may be provided about what form the paper should take and what its imagined audience would be – a mock journal article, for example, complete with abstract, introduction, and a discussion of "previous work," would be an exercise in proper citation and summarization as well as an example of writing which is intended for an audience that is generally familiar with the topic. We often accompany such assignments with sample papers illustrating the overall desired style and bibliography format, and discussions occur regarding necessary library skills to search for supporting materials and judge Internet sources with a critical eye. Where possible, papers are completed in stages to give students as much feedback as possible.

Writing across the Levels

This section illustrates more specifically how the types of writing assignments discussed previously are used throughout our curriculum.

The First Year

In the first year, all Computer Science majors must take our two gateway courses, Programming Fundamentals (COMP 115) and Data Structures (COMP 116). Since the focus of these is to develop basic programming skills, the use of whole-program documentation (Summaries) and documentation within the program (Preconditions, Postconditions, and Invariants) is introduced early, modeled extensively by both the professors and the textbook, and is a central, required part of all programming assignments (typically, six per semester). As assignments become more complex and multiple files may need to be organized, the README is introduced as a way to inform the professor about the distribution of the project across the files as well as to tell him/her how to assemble the project in order to run the program and grade it.

In the Spring semester of the first year, Computer Science majors are also encouraged to take the first of the more theoretical courses, Foundations of Computing Theory (COMP 111). In this course, they first encounter the writing of mathematical proofs.

Beginning in year one and throughout the curriculum, faculty expose students to functional and detailed specifications with each programming assignment. If an ambiguous statement appears in the specification (i.e., the directions for the current programming assignment), students often notice once they begin coding a solution. Each such ambiguity in the description creates a teachable moment on the critical need for precise descriptions.

Mid- and Upper-Level Computer Science

Students who continue with the Computer Science major start with a course on Algorithms (COMP 215), which combines proof writing with more extended programming assignments (and their accompanying documentation). Majors also must take Mathematics courses to complete the major, including Discrete Math, which includes a significant focus on writing mathematical proofs.

From the third semester onward, majors proceed to start taking upper-level courses. Courses such as Artificial Intelligence (COMP 255) and Theory of Computation (COMP 375) continue the exercise of proof-writing. In all upper-level courses, the use of program documentation continues to be a central part of regular programming assignments. Computer Organization (COMP 220) has a research paper on current hardware technologies.

We are in the process of introducing more “traditional” writing assignments, i.e. the research paper, into more and more of our 300-level courses.

Senior Capstone Experience

Our senior Capstone is a Software Engineering course, in which students experience the multiple stages of a Software Engineering project, each of which is associated with its own writing process:

- The project must be proposed to the professor and refined using drafts of a Functional Specification.
- During the design phase, a complete Design Specification is crafted.
- When the students write the actual program code, they must continue to exercise the internal and external documentation in the form of Program Summaries, Pre/Post-Conditions, and Invariants.
- Before submitting the final result, the students must complete all of the essential documentation for the “end user,” including a complete and polished User Manual.

Senior Thesis

Our best seniors complete a senior thesis. This often incorporates a significant programming project (in which case the usual components of documentation are an integral part), and in all cases involves the production of a significant, coherent, and documented piece of writing work.