An Assessment of the Writing of Undergraduate Computer Science Students

Tom Nurkkala &
Maria Gini

Department of Computer Science
University of Minnesota

A research grant report submitted to the Center for Interdisciplinary Studies of Writing

Technical Report Series
No. 20 ♦ 2002

January 2002
Lillian Bridwell-Bowles, Series Editor

An Assessment of the Writing of Undergraduate Computer Science Students

Tom Nurkkala & Maria Gini

**Department of Computer Science
University of Minnesota**

A research grant report submitted to the Center for Interdisciplinary Studies of Writing

**Technical Report Series
No. 20 ♦ 2002**

**January 2002
Lillian Bridwell-Bowles, Series Editor
Mesut Akdere, Editor**

THE CENTER FOR INTERDISCIPLINARY STUDIES OF WRITING
UNIVERSITY OF MINNESOTA
227 LIND HALL
207 CHURCH STREET S.E.
MINNEAPOLIS, MN 55455

Director:	Lillian Bridwell-Bowles, Professor of English
Assistant to the Director:	Ann Browning
Research Assistants:	Mesut Akdere, Linda Tetzlaff, Erin Harley, Susan Leem, Sara Berrey

Policy Board: Thomas Augst, Assistant Professor, English; Lisa Borys, Graduate Student, English; Terence Collins, Professor, General College; Darwin Hendel, Assistant Professor, Educational Policy and Administration; Lee-Ann Kastman Breuch, Assistant Professor, Rhetoric; Elizabeth Leer, Graduate Student, Curriculum and Instruction; Holly Littlefield, Teaching Specialist, Carlson School of Management; Carol Miller, Associate Professor, American Studies; Robin Murie, Program Director, General College; Rosemarie Park, Associate Professor, Education; Nora Paul, Program Director, School of Journalism/Mass Communication; Jim Perry, Professor, Forest Resources; Tom Reynolds, Assistant Professor, General College; Don Ross, Professor, English; Geoffrey Sirc, Associate Professor, General College; Pat McCarty Veach, Professor, Educational Psychology; Art Walzer, Associate Professor, Rhetoric

Copyright © 2002 by The Board of Regents, University of Minnesota
All Rights Reserved

ISBN: 1-881221-45-8

The University of Minnesota is committed to the policy that all personnel shall have equal access to its programs, facilities, and employment without regard to race, color, creed, religion, national origin, sex, age, marital status, disability, public assistance status, veteran status, or sexual orientation.

Preface

The Center for Interdisciplinary Studies of Writing offers research grants that have the potential to contribute knowledge about academic literacy in six areas: (1) curricular reform through writing-intensive instruction, (2) characteristics of writing across the curriculum, (3) connections between writing and learning in all fields, (4) characteristics of writing beyond the academy, (5) effects of ethnicity, class, and gender on writing, and (6) the status of writing ability during the college years.

In 1992 the Center awarded Tom Nurkkala and Dr. Maria Gini a grant for a project entitled "An Assessment of the Writing of Undergraduate Computer Science Students" Their research study was in the Center's category of "connections between writing and learning" and investigated the writing competencies of the Computer Science students at the University of Minnesota during the 1991-1992 academic year.

Tom Nurkkala was a Ph.D. student in Computer Science program at the time of the study and received his doctorate in 1996 from the University of Minnesota. He received his bachelor's degree from Michigan Technological University, and his master's degree from the University of Minnesota. Dr. Maria L. Gini is a Morse-Alumni Distinguished Teaching Professor of Computer Science. Her research interests center around intelligent agents, in particular self-interested agents that achieve their tasks by engaging in negotiations with other agents, and robotic agents that operate in unstructured and only partially known environments. Examples of research projects include: (1) building systems made up of many small robots for tasks such as exploration and mapping that require cooperation and coordination; (2) creating robots that learn to perform tasks that require complex sensory-motor coordination such as learning to backup a tractor-trailer; (3) designing software agents for tasks such as finding and organizing information on the Web, or negotiating with other agents in electronic commerce applications.

We believe that this study will provide new insights for teachers and researchers in the field of Computer Science. We invite you to contact the Center about this publication or any others in the series. We also appreciate comments on our publications.

Lillian Bridwell-Bowles, Series Editor
Mesut Akdere, Editor
February 2002

Introduction

Professional computer scientists are regularly called upon to write technical documents that specify or describe various aspects of their work. One important type of writing is external documentation, which details the design choices, significant algorithms, important structures, functional characteristics, and implementation details of a computer program. In contrast to comments scattered throughout the program code itself, external documentation is written as a coherent document in its own right, and is crucial for the program's ongoing development, support, and utilization.

Undergraduate programming courses typically require documentation for programming assignments, but assessing its quality often falls victim to the exigencies of teaching load and an understandable focus on the program itself. Thus, Computer Science students are ill prepared for professional responsibilities that require writing competence. Our study shows the need for significant improvement in student writing if we are to prepare students for the demands of writing in their profession. As a step toward improving writing quality, we present a model for writing instruction in computer science classes that makes use of a student 's knowledge about programming to improve his or her writing.

The remainder of this paper is organized as follows. Section 2 presents the major results from our study of undergraduate student writing. Details are in Appendix A. Section 3 suggests a instructional strategies designed to remedy the writing problems identified in Section 2. A survey of some computational tools for writing analysis appears in Section 4. Section 5 presents related work in technical writing and writing about computer programs. Topics for future research are considered in Section 6.

Writing Study

Method or Study

We analyzed documentation written by several groups of students, and identified common factors in the writing that lead to low-quality documentation. To obtain student-written documentation, Dr. Gini required all students in classes she taught during the 1991-92 academic year to write a short paper describing each of the programs they submitted for class assignments. This included 1 or 2 programs for CSci 5511 (Day and Extension) and 4 programs for CSci 3317 (Day and Extension). The total number of students taking the classes 2 WRITING STUDY 5 was 190; class distribution is shown in Table 1.[1] Papers vary in length from less than a page to several pages, with shorter papers typically submitted by undergraduates in CSci 3317.

Class	Students
CSci 5511 Day	74
CSci 5511 Ext	36
CSci 3317 Day	51
CSci 3317 Ext	29

Table 1: Distribution of Students in Classes Studied

Since many of the students in CSci 5511 were graduate students, and many of those in the Extension classes were part-time students, often with many years of programming experience, the bulk of our analysis focused on writing done in the CSci 3317 day classes. Specifically, we analyzed 28 randomly selected documents written by

[1] CSci 5511 Day includes 24 UNITE and NTU students.

CSci 3317 students, and considered 10 documents randomly selected from those 28 in particular detail.

Summary of Results

This section presents a summary the good, the bad, and the ugly of the writing samples from CSci 3317. While our focus is on the writing done in CSci 3317, an informal reading of the papers from CSci 5511 and the Extension classes showed:

1. The writing of Extension students was best in terms of organization and clarity as compared with all other student-written documentation we read.
2. Writing by CSci 5511 students was poorer than that of Extension students but better than that of undergraduates in CSci 3317.

The Good

While not in the majority, a few of the sample documents were quite well written. In particular, the following ideas made for particularly readable and informative documentation.

1. Some students wrote their documentation as a narrative of the steps they used in creating the program. This provided good insight into the problem solving process, and nicely complimented the program listing.
2. Some students chose to structure their documentation as a high-level summary of the main algorithm implemented in the program. This is not always an appropriate strategy, as it sometimes becomes a restatement of information that is already obvious from the program itself. However, some samples made use of this format to provide additional conceptual material that justifies and expands on the algorithm used.

3. A few students covered the theoretical considerations of the program in more detail. While such exposition probably seems redundant to students in the context of a class assignment, this kind of background information will be required in documents written by students in industry.
4. One student in the sample relied on a straightforward outline format to write a particularly lucid document. It clearly described the data structures and algorithms employed in the program.
5. A few students made uncommonly good use of typographical conventions to set off names of procedures and data structures from prose in the documentation.

The Bad and Ugly

In contrast to the previous section, the bulk of student writing is replete with errors. This section presents a brief overview of problems identified in the sample writing. Details are in Appendix A. We identified errors in the following major categories:

1. Punctuation: misuse of punctuation, poor spelling and capitalization, poor typographical conventions.
2. Sentence Construction: run-on sentences, sentence fragments, ungrammatical constructions.
3. Paragraph Construction: poor transitions, paragraphs too long or too short.
4. Verbs: passive voice, number and person errors.
5. Usage: verbose, informal, and awkward constructions.
6. Organization: poor document structure and redundant information.

We analyzed 10 randomly selected documents from our sample in particular detail. The total number of errors in each of the categories is shown in Table 2. For each category, the percentage of all errors is also shown. These data show that the bulk of writing errors in student documentation appear in punctuation and use of verbs, with a significant number of usage errors.

For each of these same 10 documents, Table 3 shows the sample number, score given to the assignment, length of document in words, number of errors of all types in the document, and the number of errors per 100 words. This table shows quite clearly that writing errors are not being used effectively as a criterion for assigning grades. Note that the table is ordered by score, and that there appears to be no correlation between the error rate (last column) and the score assigned. In addition to the 10 papers for which we made a detailed count of errors by category, we analyzed 18 additional documents, noting those categories in which at least one error was present. A summary of error frequency in all 28 papers appears in Table 4. This table has the same format as Table 2, but shows the number of papers with errors in each category. This larger sample corroborates our detailed data, suggesting that the major problem areas in our sample are punctuation, verbs, and usage.

Writing Improvement

This section describes some instructional strategies we think should be effective in addressing the problems areas identified in Section 2.2.2.

Supply a Clear Framework

When asked to write external documentation for their programming assignments, many students were unsure of the purpose or content of such a document. We feel the following issues must be emphasized in order to provide students with a coherent conceptual framework for effective writing.

1. Program documentation should explicate the thought processes and design decisions that went into creating the program code. That is, its role is to answer questions about why the program is as it is, rather than how the program is implemented. In our writing samples, many students simply restated information that was clear from the program code. Thus, the external documentation provided no more information than did reading the program itself.
2. Algorithms and data structures can best be viewed as active agents in documentation. When speaking about programs, computer scientists frequently anthropomorphize their programs (e.g., "Procedure IYZ talks to procedure J.BC and asks for the results.") However, this notion of active objects is lost when translated into written form. While such verbiage should not appear verbatim in written documentation, the spirit of active agents should be retained. Such writing will eliminate most problems with usage of verbs, since many of the errors result from viewing software as passive rather than active.

Provide Samples of Good Writing

There is certainly no substitute for providing students with positive examples to follow. In one of Dr. Gini's classes, we distributed sample external documentation for a quite complex program. This document was used as a springboard for discussion, and was covered in detail in her class. It appears students benefited from even the few ideas and guidelines brought out by the sample.

The sample document appears in Appendix B. The footnotes supplied with the sample were provided as commentary on proper form and writing style for good program documentation. These will be of most interest to readers of this paper.

Focus on the Impact of Good Writing

Dr. Gini informed her students that their writing would be used in this research project. Knowing this fact gave them additional enthusiasm and interest in writing. Although this study extended over only a single academic year, it seems clear that simply bringing the importance of high quality writing to students' attention can lead to improved performance.

Analogize Good Programming and Good Writing

We believe many of the writing problems we have identified in this study can be addressed by appealing to a "programming paradigm" for writing. That is, by encouraging students to write better documentation by making use of some of the same skills they have already acquired for writing good programs.

For example, students learn early in their programming work that attention to detail is crucial-a misplaced semicolon or comma can be disastrous. Applied to writing, this same emphasis should help eliminate punctuation errors. Similarly, students are taught that a key to writing a complex program is a "divide and conquer" approach that requires the problem be broken down in to manageable parts and the solutions combined to solve the larger problem. In short, they are taught to organize their programming. Obviously, these same skills could be called into action to improve the organization of their writing.

Computational Tools

We hoped this study would employ computerized tools to aid the process of assessing student writing. With the advent of such commercial writing analysis programs as *Right Writer and Grammatik*, we assumed much of this process could have been automated. However, after evaluating several of these programs, we concluded that they were not suitable for our assessment task.

The current generation of writing analysis software is quite primitive. Stratton [42] and Oliver [36] point out that they focus principally on the surface structure of sentences, or use various simple tricks to appear to have greater analytical power (e.g., to identify passive verbs). All the commercially available packages are limited linguistically, unable to perform the kind of sophisticated syntactic or semantic analysis necessary to make such software broadly useful. Furthermore, in order to be useful in analyzing a particular type of writing (e.g., software external documentation), these programs would require additional capabilities from Artificial Intelligence, such as detailed domain knowledge about the writer, writing process, software documentation in general, and reasoning capabilities to make use of this knowledge.

Another possible avenue for integrating computers into the writing process is Computer-Support Cooperative Work (CSCW),[2] which makes use of such computational tools as electronic mail and conferencing to permit fruitful group cooperation on writing and other projects. When used in this way, the computer becomes a kind of electronic amanuensis. This new role for using computers in such tasks is introduced by Winograd [47) and presented with its philosophical underpinnings in Winograd and Flores [48].

Related Work

[2] This idea was brought to our attention by Ron Zacharski, Department of Linguistics.

Many of the writing errors we found in our sample data are covered in such general writing texts as Strunk and White [45], Roman [40], and Zinsser [51]. Good general texts on technical writing and editing are Bly and Blake [4], Bolsky [6] , Burnett [9] , Brunner et al [8] , Mancuso [31], Rude [41], Stuart [43], and Young [50]. Mole specifically, works that focus on writing computer software documentation include Chisholm [11], Grimm [18], Halse [20], Johnson [26], McGehee [33], Penrose and Seiford [37), and Raven [38].

General texts on computers and writing (often focusing on the use of computers in introductory composition) include Barrett [3], Brit ton. This idea was brought to our attention by Ron Zacharski, Department of Linguistics and Glynn [7], Holdstein [24], Holdstein and Selfe [25], Montague [34], and Williams and Holt [46].

Landau [28] introduces a number of commercially available software packages for writing analysis. Details on IBM's EPISTLE text-critiquing system are reported in Heidorn et al. [23]. Chang [10] outlines his computer-based assistant for preparing government documents. Harrington and Green [21] discuss using automated tools for composition. See also Markel [32].

General Artificial Intelligence and knowledge representation for natural language are introduced by Rich [39], Winston [49], and Davis [12]. More detail on natural language processing can be found in Allen [1], Gazdar and Mellish [15, 16], Winograd [48], Harris [22], Bolc [5], and Studer [44]. Sources written more from the perspective of Computational Linguistics than Artificial Intelligence include Grishman [19], Alshawi [2], and Nijholt [35].

Related work in Computer-Supported Cooperative Work is written of in Flores et al. [13, 14], Germann [17), Lai et al [27), and Malone et al. [30, 29].

Future Research

Having identified specific problems in Computer Science student writing, future work should investigate and implement mechanisms for improving it. In addition to implementing the strategies outlined in Section 3, other areas for ongoing work include:

1. Develop writing exercises for Computer Science students appropriate to course content, either as stand-alone activities (e.g., writing postings to an electronic “bulletin board” about a program or related topic), or as a component of a larger project (e.g., more detailed external documentation, user-oriented documentation).
2. Facilitate training for Teaching Assistants involved in Computer Science classes. Since much of the writing done by students is only seen by Teaching Assistants, it is of crucial importance that they understand the importance of writing in the Computer Sciences. Furthermore, they must have the skills to evaluate student writing and suggest improvements as part of their grading activities.
3. Identify and verify better models for evaluating student writing, in order to quantify the impact of an emphasis on high quality writing.

Detailed Results

This appendix presents detailed results of our analysis of errors in student writing. We studied 28 sample documents varying in length from 104 to 1664 words. The average length of a document was 553 words. Scores given to students on the assignments for which the documents were submitted varied from 35 points to 100 points out of a possible 100 points. The average score was 91.1 points.

Errors were tallied according to the following categories. Abbreviations for the categories used in the following tables appear here in square brackets.

1. Punctuation
   a) Misuse of the comma [,], colon and semicolon [:;], hyphen [-], dash [-], and apostrophe for contraction [‘].
   b) Capitalization errors [C).
   c) Spelling errors [S]. This includes both outright misspellings, that could easily be eliminated by a spelling checker, as well as use of the wrong homonym or wrong word altogether.
   d) Poor typographical conventions [T]. This refers to failure to set off the names of program constructs (e.g., using italics, all upper case, quotes, or underlining). Without such conventions, documentation is difficult to read, since many of the names of program functions are verbs or verb phrases.
2. Sentence Construction
   a) Run-on sentences [RO], sentence fragments [FG], dangling prepositions [DP], and poor parallelism [PR].
   b) Ungrammatical constructions [UG]. This covers a myriad cases, most commonly poor subject-verb agreement. Many (but certainly not all) of the ungrammatical sentences were penned by students who are probably not native speakers of English.
3. Paragraph Construction
   a) Poor paragraph transitions [XT].

    b) Paragraphs too long [LG] or too short [ST]. This is often the result of describing either a single function in a one-sentence paragraph, or of collecting too many loosely related function descriptions into a single paragraph.
    c) Misuse of headings [HD].

4. Verbs
    a) Passive voice [PV]. Students almost uniformly write program documentation in the passive voice. Thus, students write, "The average is computed by procedure IYZ." Curiously, if students describe themselves or the reader (rather than the program) as acting, they usually use active verbs (see 4c).
    b) Tense [TN]. Tense is frequently confused, apparently because documentation covers various times in the life cycle of a program: when it is conceived, written, and run. This leads to frequent shifts in verb tense, sometimes in the same sentence.
    c) Person [PR]. Students appear hesitant to write about software objects doing anything. Instead, they describe either themselves or the reader as the actor in a sentence. For example, rather than writing, "Procedure IYZ calculates the average," they write, "I calculate the average," or "you calculate the average." Some writing switches from first to second to third person, sometimes in the same sentence.
5. Usage
    a) Verbosity [VB]. Students include plenty of dead wood in their writing.

b) Informality [IF]. While some software documentation could benefit from a less formal style, student writing sometimes takes informality to extreme. The clearest example in the sample data is one write-up that began:

> First of all, let me say that this was the most perplexing program I have ever done. Not only did it keep me up 'til the wee hours of the morning, but I had a severe urge to throw my PC off the Washington A venue Bridge.

No doubt the student should have done just that.

c) Awkward constructions [AWK]. Student writing is replete with disfigured sentences. Often this results from framing a sentence in the passive voice.

6. Organization

a) Poor organization [ORG]. Almost all sample data provided no clear overview of the program or the documentation. Students immediately dove into descriptions of minutia in the program. Some writing refers to functions or data structures before defining them, if they are defined at all.

b) Redundancy [RED]. Students often duplicate information that is clear from the program itself, rather than supplying background, theory, or motivation for the program.

Tables 5 and 6 show detailed information on the occurrence of errors in ten randomly selected papers in our sample set. The first four columns in Table 5 are:

1. Sample number [#].
2. Whether the writer was a native speaker of English [NS].

3. The score the writer received on the assignment for which the document was written [SCR].
4. The approximate length of the document in words [LEN].

The remaining columns in Tables 5 and 6 correspond to the areas enumerated above. Table 6 repeats the sample number to allow the tables to be correlated.

Tables 7 and 8 show a summary of errors found in a sample of 28 randomly selected papers from our data set (including those papers tabulated in Tables 5 and 6). Entries show the number of papers that contained at least one error of the type indicated.

Documentation Sample

This appendix contains a sample document distributed to students in CSci 3317. Its primary purposes were to:

1. Provide students with an example of clearly written external documentation.
2. Supply a set of suggestions on how to prepare such a document. These suggestions appear as footnotes interspersed throughout the text. The suggestions found there were designed to directly address problems we had identified in student writing prior to distributing this sample.

The program students were to write and document simulates a queuing system. An excerpt from the assignment gives the general nature of the problem:

> You are to implement a program to simulate a queuing system with customers waiting in line to obtain a service that is provided by anyone of a number of identical servers. You will model a setup in which there is a single queue of customers and two servers. As an example of this, consider a bank or a post office that has a single queue of customers and a number of teller windows.

The remainder of this appendix consists of the document distributed to students.

Waiting-Line Simulation June 15, 1992
Internal Documentation

IMPORTANT NOTE: This document includes comments on proper style and organization for program documentation. ID order to refer to specific items in the document, comments appear as footnotes; read them!

Introduction

This document describes the internal functionality of Assignment 3, a waiting-line simulation. It first discusses the basic data structures used by the simulation-queues and the agenda-and then covers the simulation itself.[3]

The program simulates a line of *customers*, waiting to be served by one of two servers. For example, in a post office, the customers would be those waiting in line to mail a package, and the servers would be postal clerks behind the counter.[4]

Queues

A queue is a linear sequence of data elements. Elements are added at one end (the "rear") and removed from the other (the "front"). Thus, data elements are stored in a "first-in, first-out" order. The program uses queues for two purposes:[5]

1. To represent the line of customers waiting for service.

2. To store events on the agenda that are scheduled to occur at the same time.

[3] Be sure to give your reader a top-down perspective on the organization of the document. This makes it easy to locate important information. Divide your document into sections and subsections to reflect its organization. Your readers will get the most information out of your document if they can digest it in small chunks—make paragraphs short, and divide long sections of the document into shorter subsection.
[4] Use concrete examples to clarify abstract ideas in the program. Your documentation is providing a link between a real-world problem domain and a computational abstraction. Make this link as clear as possible.
[5] Don't just describe your data objects—say what they are used for in the program. Again, be sure to connect the objects in the program to the real-world problem.

The queue used in the program is from Abelson and Sussman, pp. 208-212. It is implemented as a list of the data elements in the queue, augmented with front and rear pointers to improve efficiency when adding new data elements.

The underlying representation of the queue is implemented with four functions. The functions **front-ptr** and **rear-ptr** *return* the queue's front and rear pointers, and corresponding functions **set-front-ptr!** and **set-rear-ptr!** set the pointers.[6]

The basic queue operations are implemented by the following functions:[7]

1. **empty-queue?** returns **#T** if the queue is empty.
2. **make-queue** returns a new, empty queue.[8]
3. front returns the first element in the queue. It does not modify the queue (**delete-queue!** does that job**). front** raises an error if the queue is empty.
4. **insert-queue!** adds a new data element to the end of the queue.
5. **delete-queue!** removes the first element of the queue. It causes an error if the queue is empty.
6. **write-queue** writes the contents of the queue to standard output.

The Agenda

The agenda stores events according to the time at which they are to occur. Again using the post office example, the agenda is used by the simulation in two ways:

[6] Use some consistent typographical convention to set off the names of functions and data objects in your program. If you use A word processor or typesetting program, choose a different font (as in this document). Otherwise, use some other convention, like enclosing function and data names in quotes.

[7] Use lists to present information that is natural to enumerate, e.g., related functions, data objects, and messages. A list makes it straightforward for your reader to locate information by dividing your description into easily recognized and logically related components.

[8] A function in your program is an active entity-it does things. When describing what it does, use the active rather than the passive voice. Write "make-queue returns a new queue," father than "A new queue is returned by make-queue." Passive voice makes your writing harder to understand, and laborious to read.

1. When a customer is served by the postal clerk, the clerk will complete the customer's request after a particular interval of time. The agenda is used to keep track of when the clerk will be finished serving the customer.
2. The agenda stores the next time at which a clerk should check if a new customer has arrived at the post office. Think of this as the next time the clerk should look to see if there are customers waiting.

The agenda used in the program is from Abelson and Sussman, pp. 226-230. Rather than rely on "wall clock" time; the agenda uses a simple integer value to indicate the time an event is to occur. Paired with this time is a queue containing events that take place at that time. This structure allows the agenda to keep track of several events that are scheduled at the same time-each event is placed on the proper queue (**with insert-queue!**). Together, the time and the queue form a segment. The agenda is stored as a list of segments, ordered by their times. The following diagram represents the contents of the agenda (the symbol "***agenda***" is a dummy list member that tags the list as an agenda):[9]

```
(*agenda*
        (time1 (queue1))
        (time2 (queue2))
                …
        (timeN (queueN)))
```

When the simulation retrieves data elements from the agenda, it first gets each event (in queue order) from the first queue, then each event from the second queue, and so on.

[9] A picture, diagram, or fragment of code can be much more helpful to your reader than a lengthily and convoluted description.

The underlying representation of the agenda is implemented by the following functions:[10]

1. **make-agenda** returns an empty agenda. It contains a single segment with time = 0, and an empty queue.
2. **make-time-segment** makes a single segment from a time and a queue. segment-time and segment-queue return the time and queue members of a segment.
3. **segments** returns the list of all segments in the agenda. first-segment returns the first segment, and rest-segments returns all segments but the first. The latter two functions are used to "cdr down" the list of segments.
4. **set-segments!** updates the list of segments in an agenda.

The basic agenda operations are implemented by the following functions:

1. **current-time** returns the agenda's current time. For this implementation, the current time is the time of the first segment in the agenda's list of segments.
2. **empty-agenda?** returns **#T** if the agenda is empty, i.e., it contains no pending events.
3. **add-to-agenda!** adds a new event to the agenda at a particular time. This functions uses **add-to-segments!** to do most of its work. **add-to-segments!** recursively traverses the list of segments, looking for one whose time matches that of the event to be added. If it finds one, it adds the event to the segment's queue (using insert-queue!). Otherwise, it creates a new segment whose queue contains only the new event. Note that when **add-to-segments!** creates a new segment, it puts it in the segment list in such a way as to keep the list ordered by the time of each segment.

[10] It is usually helpful to organize the description of a complex data structure into several coherent pieces. For example, this list describes the "underlying representation" (the internal structure of the agenda), and the following list enumerates "basic operations" (the interface to the agenda used by other parts of the program).

4. **insert-new-time!** puts a new segment in the agenda. It is only used by **add-to-agenda!**.

5. **first-agenda-item** returns the first item on the agenda. The first item on the agenda is the front of the queue in the first segment.

6. **remove-first-agenda-item!** deletes the first agenda item from the agenda.

7. **write-agenda** writes the entire contents of the agenda to standard output.

Waiting-Line Simulation

As mentioned, the program simulates a line of *customers*, waiting to be served by one of two *servers*. The program represents each customer and server by a function. Each function is an "object" that:

* has *local state information*—it has local variables in which it can keep track of its own state of operation. (Actually, only servers have local variables; customers do not.)

* uses *message passing* to interact with other parts of the program—it acts by receiving and processing messages.

Server and customer objects are created by the make-server and make-customer functions, respectively. Each defines and returns a local dispatch function. This function handles messages sent to the server or customer. For example,[11] if aerver1 is a server object (created by **make-server**), the program would send a **write-report** message to that object by calling (**server1 'write-report**). This call invokes the dispatch function for **server1**, which in turn calls the locally-defined **write-report** function inside the server object. Locally-defined functions have access to the local variables defined when **make-server** was called (e.g., **total-service-time**).

[11] When describing a complex issue in your program, you may want to use a simplified hypothetical example. While it isn't directly relevant to the program at hand, it may make central ideas in the program clear to the reader.

Global Data

The simulation uses three global data structures:[12]

1. **the-line** is a queue of waiting customer objects.
2. **servers** is a list of server objects available to serve customers.
3. **the-agenda** is an agenda that keeps track of all the events in the simulation.

B.4.2 Customer Objects

Customer objects are created by calling make-customer. Associated with each customer are two local values:

1. **arrival** is the time at which this customer is to arrive for service.
2**. service-time** is the length of time it will take to serve the customer.

Customer objects respond to the following messages:

1. **arrival**—return the **arrival** time for this customer.
2. **how-long?**—return the **service-time** for this customer.
3. **done**—write a report of the arrival time, service interval, and departure time for this customer.

Server Objects

Servers' objects are created by calling make-server. It defines the following variables, which store local state information for each server object:

1. **customer** stores the customer currently being served by this server (if any).
2. **free** is a Boolean variable indicating whether this server is free to serve new customers.

[12] When the remainder of a section describes how data structures are used to solve a problem, describe them early and clearly.

3. **total-service-time** tracks the total amount of time this server has spent serving customers.

4. **no-of-customers** is the total number of customers this server has served.

Server objects respond to the following messages:

1. **check**—check if there are customers waiting to be served. In response to this message, **dispatch** calls the local function check-line-and-serve. **check-line-and-serve** checks if a customer is available and if so, serves the first such customer.[13] It calls **customer-available?** to see if a customer is waiting.

customer-available? checks if there is a customer in **the-line** (i.e., the queue is not empty) and that the customer's arrival time is at or before the current time of the agenda. Note that since all customers are added to the agenda at once (refer to the description of **initialize** in section B.4.4),[14] a customer with an **arrival** time earlier than that of the agenda 's current time may appear at the front of **the-line's** queue. In this case, the server must wait until the current time is the same as or later than the arrival time of that customer. It does this by putting a new event on the agenda:

(**after-delay 1 check-line-and-serve**)

This causes the simulation to invoke the **check-line-and-serve** function after another time unit has expired in the agenda. At that point, the server will again check if a customer has arrived for service.

[13] When describing a complex function, give an overview before launching into a detailed description of what it does, or which functions it calls.
[14] When you refer to a function or data object you haven't described, provide a pointer to where the information appears.

When **customer-available?** returns **#T**, the server calls the **serve** function to simulate serving the next customer. **serve** retrieves the next customer from the front of **the-line**. It up-dates its local variable **customer** to store the customer object it is serving, and sets its **free?** variable to **#F** to indicate it is busy. **serve** deletes the customer from the-line, and adds a new event to the agenda:

(**after-delay (customer 'how-long?) service-completed**)

This call arranges for the simulation to invoke the **service-completed** function after the time required to service this customer (retrieved by sending the **how-long?** message to the customer object) has expired. The serve function then updates its local variables to reflect additional service time, and an additional customer served.

After the **service-time** for the customer has expired, the simulation calls **service-completed**. **service-completed** sends a done message to the customer object (causing it to print a report of its activity), and sets the server's **free** variable to **#T** to indicate it is available to serve new customers. Finally, it calls **check-line-and-serve** again to look for another waiting customer.[15]

2. **free?**—return whether this server is free to serve customers, or is already busy with a customer. This message simply returns the value of the local variable **free**.

3. **report**—write a report of server activity to the standard output.

The function **write-report** handles this message.

Initialization

[15] When describing the details of a function, don't just repeat what the code does. Connect the operation of the code with the process of solving the problem. For example, saying "**service-completed** sets **free** to **#T**" is obvious from the code. What the reader needs to be told is *why* it's doing this—"to indicate it is available to serve new customers."

The **initialize** function sets up the simulation. It first initializes the global data structures as described in section B.4.1, then adds customers to **the-line** by calling **insert-queue!** with the customer objects returned by calls to **make-customer**.

To get the simulation started, initialize adds some initial events to **the-agenda** at time = 0. For each of the two server objects, it adds a function call that will send the **check** message to the server. Thus, the first events in the simulation will be to get both servers to look for customers.

Simulation

The simulation itself is implemented in the **simulate** function. **Simulate** assumes that the simulation has been set up by a prior call to initialize.

simulate defines the function **all-free?**, which checks if all (here, "both") servers are free. It works by moving recursively down the servers list, sending the free message to each server. If any returns **#F**, **all-free?** returns **#F** immediately. If all servers return **#T**, **all-free?** returns **#T**.

The simulate functions is a simple loop that takes the first event off the agenda and executes it. The simulation terminates when the queue, of customers (i.e., **the-line**) is empty and all servers are free (checked by calling **all-free?**). When **simulate** determines that the simulation is over, it writes a report of each servers' activity by sending each server object the report message, and then terminates. Otherwise, on each iteration, **simulate** gets the first event off the agenda (by calling **first-agenda-item**), and executes it. Since each agenda item is a function call, the event is executed by the line "(**first-item**). This item is then removed from the agenda, and simulate loops by calling itself tail-recursively.

[1] James Allen. *Natural Language Understanding.* The Benjamin/Cummings Publishing Company, Inc., 1987.

[2] Hiyan Alshawi. *Memory and Context for Language Interpretation.* Studies in Natural Language Processing. Cambridge University Press, 1987.

[3] Edward Barrett, editor. *Text, Context, and Hypertext: Writing with and for the Computer.* The MIT Press, 1988.

[4] Robert W. Bly and Gary Blake. *Technical Writing: Structure, Standards, and Style.* McGraw-Hill Book Company, 1982.

[5] Leonard Bolc, editor. *Natural Language Parsing Systems.* Springer- Verlag, 1987.

[6] Morris I. Bolsky. *Better Scientific and Technical Writing.* Prentice Hall, 1988.

[7] Bruce K. Brit ton and Shawn M. Glynn, editors. *Computer Writing Environments: Theory, Research, and Design.* Lawrence Erlbaum Associates, 1989.

[8] Ingrid Brunner, J .C. Mathes, and Dwight W. Stevensen. *The Technician as Writer: Preparing Technical Reports.* Bobbs-Merrill Educational Publishing, Indianapolis, 1980.

[9] Rebecca. E. Burnett. *Technical Communication.* Wadsworth Publishing Company, Belmont, California, second edition, 1990.

[10] Frederick R. Chang. Revise: A computer-based writing assistant. *Journal of Technical Writing and Communication,* 17(1): 25-43, 1987.

[11] Richard M. Chisolm. Selecting metaphoric terminology for the computer industry. *Journal of Technical Writing and Communication,* 16(3): 195-220,1986.

[12] Ernest Davis. Representations of Commonsense Knowledge. Morgan Kaufmann Publishers, Inc., 1990.

[13] Fernando Flores, Michael Graves, Brad Hartfield, and Terry Winograd. Computer systems and the design of organizational interaction. *ACM Transactions on Office Information Systems,* 6(2): 153-172, 1988.

[14] Fernando Flores and Juan J. Ludlow. Doing and speaking in the office. In G. Fick and R. H. Sprague, editors, *Decision support systems: Issues and Challenges,* pages 95-118. Pergamon Press, 1980.

[15] Gerald Gazdar and Chris Mellish. *Natural Language Processing in LISP: An Introduction to Computational Linguistics.* Addison-Wesley Publish Company, 1989.

[16] Gerald Gazdar and Chris Mellish. *Natural Language Processing in PROLOG: An Introduction to Computational Linguistics.* Addison-Wesley Publish Company, 1989.

[17] Clark G. Germann. New tools for the office and classroom: Using the new generation of microcomputer-based style and grammar analyzers. In *Proceedings of the 96th International Technical Communication Conference,* pages 70-71. Society for Technical Communication, 1989.

[18] Susan J. Grimm. *How to Write Computer Manuals for Users.* Lifetime Learning Publications, Belmont, California, 1982.

[19] Ralph Grishman. *Computational Linguistics: An Introduction. Studies in Natural Language Processing.* Cambridge University Press, 1986.

[20] Ronald Halse. Computer manuals for novices: The rhetorical situation. *Journal of Technical Writing and Communication,* 16(1/2): 105-120,1986.

[21] Anne Harrington and Michael T. Green. Using automated writing tools to motivate interest in the composing process. In *Proceedings of the 96th International Technical Communication Conference,* pages 76-78, 1989.

[22] Marry Dee Harris. *Introduction to Natural Language Processing.* Reston Publishing Company, Inc., 1985.

[23] G. E. Heidorn, K. Jensen, L. A. Miller, R. J. Byrd, and M. S. Chodorow. The EPISTLE text-critiquing system. IBM Systems Journal, 21(3): 305-326, 1982.

[24] Deborah H. Holdstein. *On Composition and Computers.* Technology and the Humanities. The Modern Language Association of America, 1987.

[25] Deborah H. Holdstein and Cynthia L. Selfe, editors. *Computers and Writing: Theory, Research, Practice.* The Modern Language Association of America, 1990.

[26] Gerald J. Johnson. Agents, engines, traffic, objects and illusions: Paradigms of computer science. *Journal of Technical Writing and Communication,* 21(3): 271-283, 1991.

[27] Kum- Yew Lai, Thomas W. Malone, and Key-Chiang Yu. Object lens: a “spreadsheet” for cooperative work. *ACM Transactions on office information systems,* 6(4): 332-353,1988.

[28] Ted Landau. Proper English. *MacUser,* pages 113-124, September 1991.

[29] Thomas W. Malone, Kenneth R. Grand, Kum- Yew Lai, Ramana Rao, and David Rosenblitt. Semistructured messages are surprisingly useful for computer-supported coordination. *ACM Transactions on office automation systems,* (2): 115-131, 1987.

[30] Thomas W. Malone, Franklyn A. Turback Kenneth R. Grant, Stephen A. Brobst, and Michael D. Cohen. Intelligent information-sharing systems. *Communications of the ACM,* 30(5): 390-402, 1987.

[31] Joseph C. Mancuso. *Mastering Technical Writing.* Addison-Wesley Publishing Company, 1990.

[32] Mike Markel. The effect of the word processor and the style checker on revision in technical writing: what do we know, and what do we need to find out? *Journal of Technical Writing and Communication,* 20(4): 329-342,1990.

[33] Brad McGehee. *The Complete Guide to Writing Software User Manuals.* Writer's Digest Books, Cincinnati, Ohio, 1984.

[34] Marjorie Montague. *Computers, Cognition, and Writing Instruction.* State University of New York Press, Albany, 1990.

[35] Anton Nijholt. *Computers and Languages: Theory and Practice.* Studies in Computer Science and Artificial Intelligence. North-Holland, 1988.

[36] Lawrence J. Oliver. The case against computerized analysis of student writings. *Journal of Technical Writing and Communication,* 15(4): 309-322,1985.

[37] John M. Prenrose and Lawrence M. Seiford. Microcomputer users' preferences for software documentation: an analysis. *Journal of Technical Writing and Communication,* 18(4): 355-365, 1988.

[38] Mary Elizabeth Raven. A situational approach: Making software more functional. *Journal of Technical Writing and Communication,* 17(3): 287-301, 1987.

[39] Elaine Rich and Kevin Knight. *Artificial Intelligence.* McGraw-Hill, Inc., second edition, 1991.

[40] Kenneth Roman and Joel Raphaelson. *Writing that works.* Harper & Row Publishers, 1981.

[41] Carolyn D. Rude. *Technical Editing.* Wadsworth Publishing Company, Belmont, California, 1991.

[42] Charles R. Stratton. Anatomy of a style analyzer. *Journal of Technical Writing and Communication,* 19(2):199-134, 1989.

[43] Ann Stuart. *The Technical Writer.* Holt, Rinehart and Winston, Inc., 1988.

[44] R. Studer, editor. *Natural Language and Logic.* Lecture Notes in Artificial Intelligence. Springer- Verlag, 1990.

[45] Jr. William Strunk and E. B. White. *The Elements of Style.* Macmillian Publishing Company, third edition, 1979.

[46] Noel Williams and Patrik Bolt, editors. *Computers and Writing: Models and Tools.* Ablex Publishing Corporation, 1989.

[47] Terry Winograd. Where the action is. *Byte,* pages 256A-258, December 1988.

[48] Terry Winograd and Fernando Flores. *Understanding computers and cognition.* Ablex Publishing Corporation, 1986.

[49] Patrick Henry Winston. *Artificial Intelligence.* Addison-Wesley Publishing Company, second edition, 1984.

[50] Matt Young. *The Technical Writer's Handbook: Writing with Style and Clarity.* University Science Books, 20 Edgehill Road, Mill Valley, CA 94941, 1989.

[51] William Zinsser. *On Writing Well.* Harper &: Row Publishers, second edition, 1980.

Category	Errors	% of Total
Punctuation	81	34.6
Sentence Construction	30	12.8
Paragraph Construction	5	2.1
Verbs	64	27.4
Usage	37	15.8
Organization	17	7.3

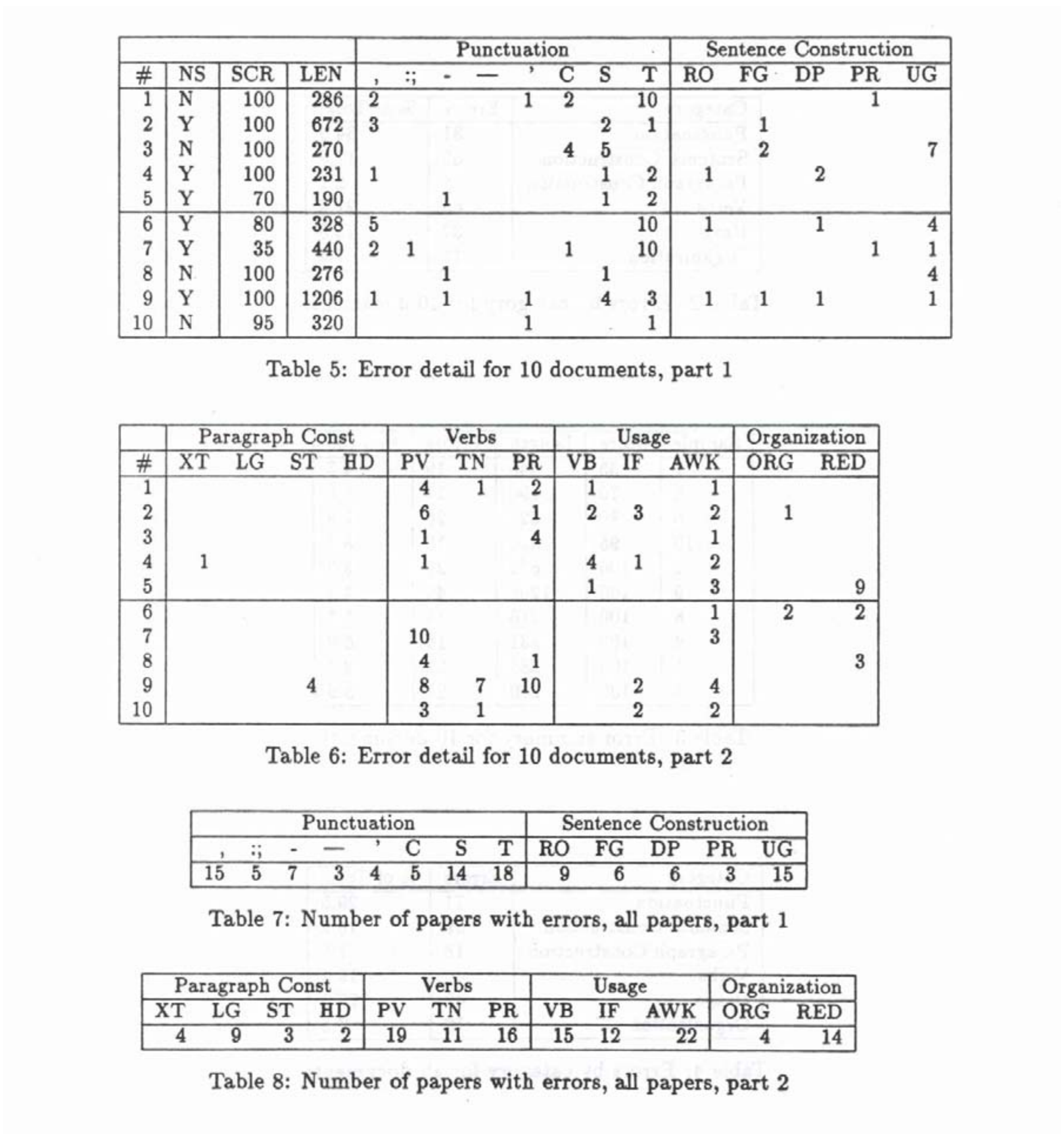
Table 2: Errors by category for 10 documents

Sample	Score	Length	Errors	Err/100
7	35	440	19	4.3
5	70	190	17	8.9
6	80	328	26	7.9
10	95	320	10	3.1
2	100	672	22	3.3
9	100	1206	49	4.1
8	100	276	14	5.1
4	100	231	16	6.9
1	100	286	25	8.7
3	100	270	24	8.9

Table 3: Error summary for 10 documents

Category	Errors	% of Total
Punctuation	71	29.5
Sentence Construction	39	16.2
Paragraph Construction	18	7.5
Verbs	46	19.1
Usage	49	20.3
Organization	18	7.5

Table 4: Errors by category for all documents



				Punctuation								Sentence Construction				
#	NS	SCR	LEN	,	:;	-	—	’	C	S	T	RO	FG	DP	PR	UG
1	N	100	286	2				1	2		10				1	
2	Y	100	672	3						2	1		1			
3	N	100	270						4	5			2			7
4	Y	100	231	1						1	2	1		2		
5	Y	70	190			1				1	2					
6	Y	80	328	5							10	1		1		4
7	Y	35	440	2	1				1		10				1	1
8	N	100	276			1				1						4
9	Y	100	1206	1		1		1		4	3	1	1	1		1
10	N	95	320					1			1					

Table 5: Error detail for 10 documents, part 1

	Paragraph Const				Verbs			Usage			Organization	
#	XT	LG	ST	HD	PV	TN	PR	VB	IF	AWK	ORG	RED
1					4	1	2	1		1		
2					6		1	2	3	2	1	
3					1		4			1		
4	1				1			4	1	2		
5								1		3		9
6										1	2	2
7					10					3		
8					4		1					3
9			4		8	7	10		2	4		
10					3	1			2	2		

Table 6: Error detail for 10 documents, part 2

Punctuation								Sentence Construction				
,	:;	-	—	’	C	S	T	RO	FG	DP	PR	UG
15	5	7	3	4	5	14	18	9	6	6	3	15

Table 7: Number of papers with errors, all papers, part 1

Paragraph Const				Verbs			Usage			Organization	
XT	LG	ST	HD	PV	TN	PR	VB	IF	AWK	ORG	RED
4	9	3	2	19	11	16	15	12	22	4	14

Table 8: Number of papers with errors, all papers, part 2