Program Design in File Structures

Susan A. Mengel, Daniel A. Tappan
University of Arkansas
Computer Systems Engineering
313 Engineering Hall
Fayetteville, AR 72701-1201
Phone: (501) 575-{5728,6036}
Fax: (501) 575-5339
E-mail: {sam,dat}@engr.engr.uark.edu

Abstract

The importance of emphasizing design in programming stems from the desire to train students in thinking more about solving the problem than programming the problem. Although sitting in front of the computer causes the student to think that he or she is getting closer to a working program with every passing minute, the student may actually spend more time on the program than necessary. More time is spent because the student may have written the program with the wrong approach and figured that out only after many hours of debugging. Further, the program may not be organized very well which also leads to long debugging periods.

To help the student streamline the development of programs, formulating a design of the program was necessary before actual code could be written in a senior level file structures course. The design component became necessary as the instructor watched students not complete the more complex B-tree and hashing programs. The design component also confirmed the correct or incorrect thought processes of the student so that incorrect notions could be corrected before programming was completed.

The design component includes drawing the file structure to be implemented and examples of the operations upon it. The design then goes into further detail by requiring the student to split the problem up into subprograms specified in pseudocode. The design process undertaken by the students is analyzed and reported in this paper as to how it helped the students to understand the file structures concepts better.

Introduction

As undergraduate students go through a computing curriculum, the programs that are assigned to the students become more complex. For example, implementing B-tree programs requires knowledge of recursion, experience in file I/O, and thoughtful planning of the structure of the data files on disk. Even though the programs may still not be so large that teams of programmers must be utilized, careful though about the programs' design on the part of the student should still be performed before writing such programs. Students, however, who are under time constraints from other courses will tend to let a program's design slide only to find the debugging task of a large program quite difficult.

It is interesting to note that design in the field of engineering is almost essential to the successful implementation of a component, such as a circuit. Indeed, powerful software packages, such as PowerView, are utilized by many schools to aid students in circuit design. For program design, software engineering tools may help, but are generally utilized in a software engineering course, as for example in [2]. Software engineering tools can have a high learning curve which is why they are generally learned in a course devoted to their use.

A problem in program design is the ability to determine if a program would actually work from the design itself. Software testing tools may be used to check program code, but not usually the design.

Even though use of a commercial design and test tool in a programming course would enhance the educational experience and give students experience with a tool used in industry, it may not be financially possible to acquire the tool or take the time to teach the use of the tool in the course. Students, however, need to practice design with or without commercial software design packages.

At the University of Arkansas in the Computer Systems Engineering Department, the file structures course is being taught with a design component on each programming assignment. The students have to turn in a design at least one week before the program is due showing their file structures, how they are manipulated, and the modules to be used for the manipulations.

File Structures Course

The file structures course is *CSEG 4553 Computer Organization and File Structure*, a senior level required course. Folk and Zoellicks' *File Structures, Second Edition* [1] book is used to give the students an in-depth look at file structures. The students study secondary storage mechanisms, such as disks, CD-ROM, and tape. They learn about file structures, like B-trees and hashing with buckets, that utilize storage mechanisms well. They also learn about external sorting and compression.

The students use the ANSI C programming language with the unbuffered file commands: `open`, `close`, `read`, `write`, and `lseek`. In general, they use PCs, but can use Sun Workstations. By the time they take this course, they will have had C, data structures, and some minor file handling experience using C buffered file commands: `fopen`, `fclose`, `fread`, `fwrite`, and `fseek`.

Four assignments were given in the spring of 1995 as listed in the next section. An assignment consisted of one or two programs, each due in succession. One week before a program was due, a design for the program had to be submitted consisting of drawings of the file structure and how it was manipulated along with the algorithms in pseudocode needed to implement the program.

Programming Assignments

File Operations

The first programming assignment was designed to familiarize the students with files and how they can be manipulated. In this assignment, two programs had to be written: one for file manipulation and one for pointer representation.

File Manipulation

The first program had the students multiply two integer matrices in two separate binary files of integers and place the result in a third binary file of integers. None of the matrices could be stored in memory; i.e., only one or two entries from each matrix could be in memory at the same time. All of the file names were user specified, non-existent files had to be created, the two

matrices to multiply could be input by the user into empty files, any entry in the two matrices to multiply could be changed by the user, and the matrices had the same file format.

The students had to do a design for the program to show how the matrices would be stored and manipulated on disk during multiplication or entry changes. It also had to show the algorithms needed for implementing the program. The algorithms included the main driver with the menu for the user, the multiplication of the two matrices on disk with the result stored on disk, and the mapping of the two-dimensional matrices onto a one-dimensional file format.

Being the first design required from the students, they naturally had only a vague idea about what was expected. Whereas by the end of the semester, the designs were generally similar between students in structure and content, this first one varied greatly. Design quality and thoroughness tended to be directly related to the students' interest in the subject and their effort to learn it.

The use of pseudocode in the design was intended to simplify the conceptual structure of the program components; i.e., it was meant to free the student from the syntactic idiosyncrasies of the programming language. Unfortunately, the students' interpretation of pseudocode often meant simply leaving the semicolon off C code.

Although most students made earnest attempts to formulate useful designs, the first diagrams were often little more than boxes drawn around vague ideas. It became important to instill in the students that the designs were for their benefit, so the better they made their designs, the easier it should be to write the actual program.

With this program, the students gained experience in opening, creating, and closing files. They worked with binary files and learned how to use `lseek` to move the file pointer to where they needed to read or write matrix entries. They also gained experience in reading from and writing to a file, as well as having more than one file open.

Pointer Representation

The second program involved implementing a linked list in a user specified binary file of employee records. The records were implemented as C structures and the list was kept in

descending order according to last name. The user could insert into, delete from, change, search, and print the linked list. For the linked list file, an available list of deleted records was maintained and a header record contained the heads of the linked list and of the available list. The linked list could not be stored in memory, but one or two entries from the list could be in memory at any one time.

The students had to do a design for the program showing how the linked list would be stored and manipulated on disk during insertions, deletions, changes, searches, and printing. It also had to show the algorithms needed for implementing the program including the main driver and menu along with the linked list operations.

For this program, the students gained more experience in file manipulations and binary files with fixed length records. They learned how to represent pointers in files as relative record numbers or as record offsets which is necessary to implement the most complex file structures that have multiple pointers per record.

This assignment presented no real difficulties to the students since nearly all of the operations were straightforward. The one major problem was that some programs did not perform correctly in maintaining the available list of empty structures. Quite possibly, maintaining two lists for the first time out in a file was considered difficult or time was a factor.

First Assignment and Class Comments

Neither of the above programs were very practical implementations of file structures. They were rather used as a means of taking familiar problems and putting them into a new setting; i.e., files. The students could concentrate more on the file implementation than on understanding an unfamiliar problem. Further, they learned how to use `lseek` to position the file pointer correctly before more difficult assignments were given and how to represent pointers in a simple linked list file structure before more complex file structures were introduced. They also gained experience in using fixed-length records.

Throughout the assignments in this class, even the best students tended to overlook small details, especially with the point values for them were low. Possibly they expected that trivial

operations, such as paging output neatly to the screen, would not be tested during grading. Moreover, many times these operations were not even considered in the designs, which may indicate that the students never even intended to implement them.

Indexing and External Sorting

The second assignment had two programs where the second program extended the first one. This assignment was considered to be the longest one by the students and involved implementing a small bibliographic database.

Bibliographic Database

The first program implemented a user specified ASCII file where the records were of variable length. Each record could be a book or journal entry. The records were in entry-sequenced order and could be inserted, deleted, changed, searched for, or printed. An available list of empty records was kept in ascending order of size for a best-fit placement strategy. The file could not be read into memory in its entirety.

The design for the first program had to show the variable length record manipulations. The students showed pictorially the insertions, deletions, changes, searches, and printing. They then showed the algorithms for the record manipulations and for the best-fit placement strategy.

Although by this point in the semester, the students had established a relatively consistent design format for themselves, they apparently did not think to create a design for more than just the internal components of the program. Because of the many menu options and input fields, nearly all programs exhibited disorganized arrangements of the elements. No one seemed to consider that these could be designed in advance.

The students learned how to work with variable length records and the problems that can occur in reclaiming space in files with such records. They had to be able to move the location of a changed record should it have increased in size after the change and reclaim its old space in the file. They also learned how to have two different types of records in the same file.

The program itself was quite long and in-depth, but most of its components were similar. Once students determined how to store and manipulate data for one database record type, it was often

a trivial matter to modify it for another. The difficulty came in implementing the operations on the different record types. Few students managed to complete all the requirements of the assignment.

Indexing and Sorting

The second program had the students add a variable length notes field to each bibliographic entry and three array indexes for faster searching: author index, title index, and subject index. Notes were stored in the same file as the bibliographic entries and could be inserted, deleted, changed, or printed. If searching was done via the subject index, then the Booleans, AND, OR, and NOT, could be used. The result of any search had to be presented in descending date order using an external merge sort with three record runs and a one-step merge. The indexes could be stored in memory, but had to be returned to a file before the program ended.

The design consisted of drawings of the file manipulations for the notes and searches, indexes and their storage representations, and the external merge sorting. The algorithms specified how to store the notes and retrieve them, how to utilize the indexes in a search, and how to perform the external merge sorting.

The students gained some experience in storing different objects in the same file: a book entry, a journal article entry, and notes. The students learned how to use indexes to allow more powerful and efficient file searches than sequential searching allows. They worked with external merge sorting on a small scale which gave them an understanding of how large files would be sorted.

Those students who had completed the first part of the assignment seemed to finish the second part relatively easily. The appeared to have had more time for the second part than those who still did not have a completely working first part.

The most common difficulty encountered in this assignment was with the Boolean operators. The initial designs made very little mention of how these would be implemented. As it turned out, many students did not get the operators working at all. Perhaps they took it for granted that the coding would be trivial.

B-Trees

Once again, the students did two programs where the second built off the first. The students had to implement a B-tree which could be used as an index into another file. Given that the B-tree is rather complex, the students worked only with it and could work together in teams of two.

Insertion

The first program had the students implement B-tree insertion where the keys to be stored were 4-byte integers. The B-tree was stored in a user specified file with a header record that contained the location of the root, the available head, and the number of integers in the file. *N* randomly generated integers, single integers, or an ASCII file of integers could be inserted into the B-tree. In addition to insertion, searching and printing the integers in ascending order were allowed. Only the B-tree root could be kept in memory during program execution. Other B-tree nodes could be read into memory, but just for the duration of the current operation (insertion, deletion, etc.). The students experimented with different B-tree node sizes 10, 100, and 1000 integers in combination with data of 500, 5000, and 10,000 integers. They wrote a report on their findings.

The design of the program had to delineate B-tree insertion, searching, and printing. The students had to show how splitting could occur on insertion and how levels could be added to the B-tree. The algorithms the students wrote outlined B-tree insertion, searching, and printing using recursion.

The students gained experience in working with a rather complex file structure. Each node had more than one pointer and recursion was used during the insertion, searching, and printing processes. They were able to experiment with different B-tree node sizes.

Although B-trees were new to most, if not all, students in the class, they found this assignment to be one of the easiest and least time-consuming. One plausible explanation is that the program had relatively few different components to it. Whereas most of the other assignments required many different operations on the file structures, here mainly only searching and inserting were necessary.

Deletion

The second program extended the first one to include B-tree deletion and deleted node reclamation through an available list. Single integers or ASCII files of integers could be input for deletion.

The design included diagrams of B-tree deletion. Deletion could involve borrowing keys on underflow, compacting nodes, and deleting a level in the B-tree. The algorithms showed how to do B-tree deletion using recursion.

The designs for this assignment were very thorough and usually showed exactly how the B-tree structures would be implemented and manipulated. Most students showed examples of all the different combinations possible, often in a cartoon-like depiction. The effort inverted into designing the file structures appeared to have helped greatly since many of these students received perfect scores. In any case, they all finished the assignment and had completely working programs.

The students gained experience in implementing a B-tree with all associated operations: insertion, deletion, searching, and printing. Deletion was a little more difficult than insertion since some deleted keys had to be replaced with a successor and then the successor deleted from the B-tree. The students also gained experience with batch input.

Hashing

For the last assignment, the students could work in teams of two and had to implement an inventory database of parts and suppliers using hashing. Only one program was required which had one hash table for the parts and one hash table for the suppliers. The hash tables were implemented as scatter tables where a hash table entry contained a pointer to the buckets storing the part or supplier records that hash to the same address. The buckets were stored in a separate file from the hash tables and were implemented as sequence sets where records were kept in ascending order by part or supplier number. Each sequence set node held four part records or four supplier records. Empty sequence set nodes were reclaimed using an available list. Both the part and supplier records were of a fixed size. The students were held to a collision rate of under 20%, they had to allow insertions or deletions from ASCII files of part or supplier records,

and they had to allow record changes and searchers. They wrote a paper on their hash function design justification as well as how they would allow the part or supplier records to be printed in alphabetical order by name given that hashing randomly distributes the records over the hash table.

The design demonstrated pictorially how insertions, deletions, changes, and searches would be performed upon the hash tables and sequence sets. Examples of hash function usage were given as well. The algorithms gave the detail of the operations, showing how sequence sets would be split or redistributed upon insertion and compacted upon deletion. Specification for the hash function computation was also given.

The designs for this final assignment were thorough. All students explicitly described their intended hashing schemes. Furthermore, most justified their choices with some mathematical formula that showed a collision rate below 20%. In practice, though, it was difficult to test whether their designs did in fact perform as given. The large scale of the database meant that many test data entries were necessary. Nearly as much thought went into grading the assignment as went into designing it.

Students mentioned in their reports that once they mastered the hashing function for one type of record, they found it easy to apply the function to the other. In addition, most did a performance analysis of their proposed design compared to the final product. From the design, they were able to see if the program functioned as expected. Normally students are satisfied if a program works at all, and not whether it is as efficient as they predicted. Requiring them to create a design forces them to consider how the end product should function.

From this program, the students learned how to implement a small database. They had to coordinate database operations so that if a part fell below a certain number upon a deletion, the part would be flagged for reorder and the supplier name and address written out. They had to work with their hash functions so that collisions would be kept to a lower number. In resolving the collisions, the students learned how to work with a sequence set which keeps records in a file in sorted order. From the paper they wrote, they had to consider how to change their database so that part and supplier records could be listed in alphabetical order by name. Since hashing

typically distributes records randomly over a table, printing in alphabetical order becomes difficult.

Analysis of the Programming Design

Interestingly enough, program completion or near program completion by the students is much better regardless of whether or not the design is checked in-depth. It is clear that the process of working with the file structures and algorithms on paper helps with their understanding of the concepts. For example, previous students could not finish a B-tree implementation in general, but students in the Spring 1995 semester who did a design could. In fact, the work of the students who did a design improved throughout the semester. They labored diligently on the designs and produced complete or very close to complete programs. By the end of the semester, their designs and programs showed a great deal of maturity.

In order to improve the design and judge its effectiveness, a survey was given to the students asking them about their design experience. They answered the following questions:

- How would you rate the importance of a design before implementing a program on a scale of 1 (not important) to 10 (important)?

  The average answer for this question was 8.5.

- How would you rate the designs you did for this course in helping you to implement your programs on a scale of 1 (ineffective) to 10 (effective)?

  The average answer for this question was 7.

- How would you rate the designs you did for this course in reducing the time you spend programming and debugging on the computer on a scale of 1 (ineffective) to 10 (effective)?

  The average answer for this question was 6.2. The average here is interesting since the course instructor could tell the students were performing better and even had less time to do the assignments than students in previous semesters. The students, however, do not seem to be making the connection between design and

implementation. Their inexperience with design may be a contributing factor in their inability to do so. In order to help them tie the two together, it may be necessary to use a software design tool for at least one program that could generate code from the design.

- How did the designs add to your understanding of the program assignments?

  The comments here included better visualization of the file structures and the sequence of steps needed to manipulate them, good reference while programming, helped to instill confidence in the concepts before program implementation, helped in determining file organization, and good for complex programs.

- What could be changed so the designs aid your understanding better?

  The comments here included more feedback on whether the design was efficient, a higher level design should be used to keep students from thinking in the C programming language, the designs are fine, talk about the designs in class after they are returned as a workshop so students can learn from each other, and include flowcharts to get the big picture of the program.

Future Work

The design experience should be such that the student makes the connection between design and lower implementation time. Although putting the design on paper helped the students in understanding the concepts, they still had to finish and type the C program into the computer after completing the design which took some time. They also had to debug the program and test it which took time was well. Still, as compared to students who did not do designs in past semesters, the current students did better on the programs and had cleaner implementations.

In order to help the student connect the design with lower implementation time, a design tool will be acquired. The design tool should be able to generate C code automatically and be easy enough to use that a large amount of time need not be spent on learning its functionality. Several firms offer substantial discounts to education institutions so the price of the tool should not be too high unless a network license is not obtainable and individual copies must be purchased.

One of the tools under consideration if System Architect, from Popkin Software and Systems, Inc., which supports several design paradigms and generates C code. The students will still be required to draw the file structure operations pictorially, but will perform program design using the tool. Hopefully, the tool also will keep the students from thinking too much about C when designing the program.

A design workshop will be added for each assignment. Once the students submit their designs, all students will get a chance to contribute to the discussion of the best of individual designs. The discussions should help the students having trouble with the material to see other ideas and should help the students having no difficulty with the material to reinforce their ideas. Since students will still have to implement a program separately, the discussions should not "write the program for the student." The designs will be weighted more in order to encourage participation by all students in the discussions.

Once the program is written and submitted, a small workshop will be help to see how the design helped to implement the program. The discussions in this workshop should help the students make the design and lower implementation time connection Also, students will benefit from each other's ideas.

Acknowledgements

All terms in this article that are known to be trademarks are capitalized. Sun Workstation is a registered trademark of Sun Microsystems, Inc. PowerView is a registered trademark of Viewlogic Systems, Inc.

References

[1] M.J. Folk and B. Zoellick. *File Structures, Second Edition*. Reading, MA: Addison-Wesley, 1992.

[2] W.M. Lively and M. Lease. "Using CASE Tools to Teach Software Design in an Undergraduate Software Engineering Lab", *1992 Annual Conference Proceedings ASEE, Volume I.* Washington, DC: ASEE, 1992, pp. 386-389.