Data Structures, Algorithms, and Software Engineering

Christian S. Collberg*
Lund University, Sweden

Abstract. *Traditionally, students in undergraduate computer science curricula have to wait until their third or fourth year until they are confronted with the problems which arise in the design of large programs. The rationale behind this is that programming-in-the-small has to be mastered before programming-in-the-large. In this paper we will argue that the elements of software engineering must be present at all levels and in all computer science courses and that it is feasible, with the proper tools and precautions, to assign large and complex programming assignments even at the introductory level. An experiment to this effect has been carried out in a data structures and algorithms course given at the University of Lund. A non-trivial program (a* make*-utility for Modula-2) was specified and partially modularized by the instructor and assigned piecewise to the students. The assignments trained the students in the implementation of data structures and algorithms as well as software tool design, modularization, information hiding, reuse, and large system design.*

1 Introduction

We believe that the principles of software design must permeate every part of a computer science program, and should not be confined to one upper-level course in software engineering. Every course, even ones at the introductory level, can contribute to the students' understanding of large systems - how they are designed, implemented and tested, and how they evolve with changing needs. In this sense our views coincide with the ones expressed in the ACM recommended curriculum for CS2 [5]. This paper describes how a traditional undergraduate data structures and algorithms course was given a software engineering flavor by careful choice of programming language and assignments.

2 The course

The Lund University computer science program, like most other computer science curricula, offers a course in the fundamentals of data structures and algorithms. The course

*Author's present address: Department of Computer Sciences, Lund University, S-221 00 Lund, Sweden. email: collberg@dna.lth.se

contains, among other things, sections on lists, trees, and graphs, algorithms for searching and sorting, and elements of the design and analysis of algorithms. The students come from a variety of backgrounds; some have almost two years of mathematics training as well as an introductory course in Pascal programming, others are starting their third year in a business computing program, and others are still freshmen with an introductory Pascal course as their only computing experience.

The students are supposed to spend an equivalent of ten weeks of full time work on the course. This includes twenty-four ninety-minute lectures and twelve ninety-minute labs, six to eight programming assignments, and a final.

For many years the assignments were written in Pascal, each assignment being a self-contained (small) program. A typical set of assignments might have included the following:

- An arithmetic RPN calculator using a stack.
- A polynomial calculator using linked lists to represent polynomials.
- A telephone directory using binary search trees or hash tables.
- An index generating program using an $O(n \log n)$ sorting method.
- An implementation of Dijkstra's algorithm for the Single Source Shortest Path Problem.

2.1 Objectives

The emphasis of the course is on data structures and algorithms and their implementation. Certain topics have to be included in the course material and this limits the amount of extraneous material, such as software engineering, which can be covered. The mandatory sections are:

abstract data types their definition and use.

data structures dynamic and static implementations of lists (one-way, two-way), stacks, queues, trees (k-ary, general), and graphs.

recursion implementation and elimination.

sets bit string, sorted array (binary and interpolation search), binary trees, hash tables (with chaining, and with linear, quadratic, double, and double ordered collision resolution), AVL-trees, 2-3-trees and B-trees on external memory.

sorting $O(n^2)$ algorithms, quicksort, heapsort and mergesort on external memory.

priority queues heaps.

analysis of algorithms Asymptotic complexity, analysis of iterative and recursive algorithms, solution of recurrence relations.

3 The language

Choosing a suitable implementation language for novice computer science students is not an easy task. Apart from obvious factors such as simplicity, expressiveness and suitability

for the assignments at hand, the availability of affordable and reliable compilers also play an important role. For the present course the choice was further complicated by the fact that the only prerequisite was a rudimentary knowledge of Pascal. Deciding on a language radically different from Pascal, perhaps in a different paradigm such as functional (ML) or object oriented (Eiffel), might force the students to spend more time getting acquainted with the new language than with the programming problems. An imperative language in the Pascal tradition was therefore deemed the most suitable choice.

Furthermore, it was felt that in order to teach the basic concepts of abstraction and software design the language ought to support modularization, information hiding, generics and generalized iteration. We concluded that Modula-2 would be the best, although far from optimal, choice:

- Modula-2 is in many ways similar to Pascal and should be easy for the students to learn.
- An acceptable compiler was already available.
- In Modula-2 information hiding is achieved through so called *opaque types*. This is a restricted form of information hiding since opaque types are essentially limited to pointers. Also, constants may not be hidden.
- There is no support per se in Modula-2 for generics. Generic ADTs can, however, be implemented using techniques discussed in [1, 9].
- Generators/iterators (in the style of CLU) are not available. They can, however, be simulated using Modula-2's procedure types.

Koffman [4] favors Modula-2 over Ada, PL/1, and Pascal for CS1 and CS2. Ada and PL/1 are rejected on the grounds that they are too large and complex and that compilers are not readily available for small computers. Pascal is rejected because of its lack of important software engineering features and because of its syntax, which is more complicated than that of Modula-2.

4 The project

The objective of the course was to give the students an idea of what they might encounter when they enter their career as computer professionals. Therefore, the programming project was given as a set of small, self-contained projects assigned at different times during the semester. The students were unaware of the ultimate goal of the project until presented with the last assignment. Thus, they may be thought of as having played the roles of *starting programmers*: largely ignorant of the overall structure and goal of a project, repeatedly given small, informally specified assignments, and unaware of how their contributions would eventually fit into the complete picture. But as the students reached the last assignment, they switched roles to become, in a sense, *project managers*. They were given a vaguely formulated description of a project which they should be able to design and implement with the aid of the tools they had constructed in previous assignments.

One of the major obstacles in the design of the course was to find a suitable, non-trivial assignment. It was felt that the following keywords described the criteria which had to be met:

usefulness To simulate the real world as closely as possible, the project should result in a useful product, not a toy.

tool-building It would be beneficial to the students' long-term understanding of software engineering if the product was a software tool which they could use in future class projects.

reuse To show the students the benefits of *reuse*, some general software component should be used more than once in the project.

relevance The course is *primarily* a course in data structures and algorithms. Hence, the individual assignments should be relevant to the material presented during class. Specifically, students should be trained in the areas of lists, trees, graphs, searching, and sorting.

non-irrelevance The students should, as little as possible, be required to implement functions unrelated to the course material. In other words, the assignments should not force the students to spend an inappropriate amount of time producing code which would not aid in their understanding of the problem area.

A *make*-utility [2] for Modula-2-programs was deemed a suitable project. Such a program was not available (this would take care of the *real world*-requirements), and it would require most of the data structures and algorithms on which the students needed training. Also, those parts which would require knowledge not obtained in the course, primarily a Modula-2 parser and certain system-dependent facilities, could easily be provided by the instructor.

The *make*-utility was envisioned as two separate programs, *dgen* (for *dependency generator*) and *m2make*. Basically, *dgen* would, by examining the source code of the modules making up a given program, build a directed acyclic graph on the import-export relationships between the modules and write this graph to a binary file. *m2make* would, when invoked, read this dependency graph and initiate the compilations necessary to bring the program up to date.

The input to the dependency generator is a text file, the *program description*, which contains a list of the modules used in the program. Two source files and two compiler-generated files are associated with each Modula-2 module: definition- and implementation files which are compiled into symbol- and object files, respectively. The *program description* must therefore contain references to the directories where these files reside. *program descriptions* may also define *aliases*, short forms of frequently used strings. The format of the *program description* is shown in figure 1.

It is important to remember that students in a second computer science course are novice programmers who have not yet reached the level of sophistication where they can take informal specifications of complex problems through the normal stages of design, formal specification, implementation, and testing. Careful wording of the assignments is therefore essential: the students must be guided over the worst pitfalls while, at the same time, be given enough latitude to explore different design strategies. Hence, the early assignments were specified in detail, giving only minor room for student design errors, whereas the later ones were more vaguely formulated.

```
ALIAS     home          /usr/chris
ALIAS     m2            $home$/m2
ALIAS     bin           $m2$/bin

MODULE    AliasSet
DEF       $m2$/alias
IMP       $m2$/alias
SYM       $bin$
OBJ       $bin$

MODULE    GenericBST
DEF       $m2$/bst
...       ...           ...
```

Figure 1: The layout of the *program description*. X should everywhere be replaced by the value of the alias X.

In each of the first five (out of six) assignments the students were asked to perform three tasks: to implement a generic package for a certain data structure or algorithm, to create an instantiation of the generic package, and, lastly to extend the dependency generator in some way using the newly designed packages. Students were expected to turn in each assignment complete with user-, system-, and program documentation, as well as with a simple interactive main program that would enable easy testing of the functions implemented thus far. The program and its documentation was, in other words, developed in parallel and in an incremental fashion.

The sixth assignment was an exercise in design and reuse. The students were given a brief sketch of a Modula-2 make-utility and were asked to design one using previously constructed tools.

Assignment # 1: Lists.

1. Implement a generic two-way linked list package, *GenericLists*.
2. Use *GenericLists* to implement a package *ModuleList* which stores names of modules and for each module the names of the directories in which its corresponding files reside.
3. Implement a program *dgen* which loads the contents of a *program description* text file into the *ModuleList*.

Assignment # 2: Binary Search Trees.

1. Implement a generic binary search tree package, *GenericBST*.
2. Use *GenericBST* to implement a package *AliasSet* which stores variable-value pairs. Both variables and their values are character strings.
3. Extend *dgen* to handle the definition and use of aliases in the *program description*.

Assignment # 3: Hash Tables.

1. Implement a generic hash table package, *GenericHashTables*.
2. Use *GenericHashTables* to implement a package *ModuleSet* which stores a set of module names and data about each module. This data will include the complete names of the files which belong to the module.
3. Extend *dgen* to load data into *ModuleSet* from *ModuleList*.

Assignment # 4: Graphs.

1. Implement a generic directed graph package, *GenericGraphs*, using the adjacency-list technique. Use *GenericLists* to implement the adjacency-lists.
2. Use *GenericGraphs* to implement a package *ImportGraph* where each vertex represents a file and each edge a dependency between two files.
3. Extend *dgen* in the following way:
   - Create a graph node for each file in *ModuleSet*.
   - Store references to the nodes created in the *ModuleSet* entries for the corresponding modules.
   - Build an import dependency graph for the modules in *ModuleSet*. Use *M2Parser*, a parser for Modula-2 provided by the instructor.
   - Test the graph for circularity. A circular graph represents a non-compilable Modula-2 program.

Assignment # 5: Sorting.

1. Implement a generic sorting package, *GenericSort*, using an $O(n \log n)$ algorithm.
2. Use *GenericSort* to extend *GenericHashTables* with a function to iterate over the elements in the table in sorted order.
3. Extend *dgen* to create a listing file containing:
   - A list of the modules and their corresponding files, sorted on the module names.
   - The dependency graph.
   - A list of the aliases and their values, sorted on the variable names.

Assignment # 6: A Make-utility.

1. Implement a dependency generator *dgen* for Modula-2. *dgen* should as input take a text file, the *program description*, containing a list of the modules used in a program. It should generate two files, a listing file and a binary file containing a dependency graph.
2. Implement a *make*-utility, *m2make*, for Modula-2. *m2make* should as input take a dependency graph-file for a program, traverse the graph in reverse topological order, and update the program by initializing the compilation of the modules which are out of date.

5 Software engineering issues

Although primarily an exercise in the implementation of data structures and algorithms, the *make*-project covered many, albeit not all, pertinent software engineering issues. The choice of Modula-2 as the implementation language provided a natural stepping stone for training the students in the design of low-level specifications and in the hiding of representational details of modules. The concept of *reuse* was introduced in two ways: by making all low-level abstract data types general and generic and by forcing the students to structure their programs in such a way that major parts could be used more than once. *Maintenance* is, by nature, always difficult to teach within the constraints of an educational setting. Here the students were introduced to some of the issues involved when the incorporation of new code into their program required changes to old code. In order to promote a sound attitude towards *testing* and *documentation* each assignment was regarded as a program in its own right, complete with documentation and routines for testing.

6 Student reactions and results

A survey administered to the students at the end of the course revealed some expected but also some surprising results. A majority of the students were enthusiastic about the project as such, but, as might be expected, thought the workload was too heavy. A typical project contained approximately 2300 lines of code in implementation modules and another 300 lines of code in definition modules (see table 1). One of the primary reasons for the present project design was to minimize the amount of extraneous code, i.e. code unrelated to the subject matter. According to the students this goal failed. Some claimed that as little as 10% of their total programming time was spent implementing important data structures and algorithms, and as much as 50% was spent on "mindless hacking". We feel that these figures should be taken with a grain of salt. They are most likely a reflection of the frustrations programmers often experience: more time is spent structuring and restructuring one's program into appropriate units and subunits than on code which "gets the job done".

Many students had problems with string manipulation which is virtually nonexistent in Modula-2. Only a few had caught the hints from the instructor that designing ones own string manipulation package might be a good idea. Those who did turned in cleaner and simpler solutions.

	DEFINITION	IMPLEMENTATION
Source code lines	633	3085
Source code lines (stripped)	298	2335
Number of modules	13	15
Number of procedures	138	168

Table 1: Project statistics. Numbers are per project averages. Blank lines and comments have been removed from stripped source code.

The relationship between generic modules and their instantiations also puzzled many students. Perhaps this problem can be attributed to Modula-2's limited support of generic modules and could have been alleviated if another language, such as Ada, had been used.

Abstractions were, as always, a great mystery to the students. Typically they had problems distinguishing between a *Mapping-ADT*, such as the module *AliasSet*, and its implementation, in their case a binary search tree. Complaints about the fact that "everything has to be hidden" were frequent at the beginning of the course but thinned out as the students' programs grew in size and complexity.

Students often complained that the assignments were too informal and unprecise. When the instructor pointed out that a student's program did not catch a certain error condition, the student invariably replied that that particular error was not defined in the assignment. Many students failed to see that the lack of details in the specifications of the assignments was not an oversight on the part of the instructor, but an indication that they should think for themselves and, if that was not enough to resolve some ambiguity, ask the instructor. When queried about some fairly obvious design flaw in their program, many responded that they had wanted to do a different design but did not know whether *they were allowed to*. These and other problems can in part be attributed to the students' lack of maturity and in part to the instructor's inability to explain to the students the primary objectives of the course.

All in all, students reacted very favorably to the project and preferred it to the set of many small assignments they had had in earlier courses.

7 Relation to previous work

Courses with similar objectives have been described before [3, 6, 8]. None of these courses, however, have the strong emphasis on incrementality of implementation and documentation, tool design, real world applicability, and reuse as the one described in this paper. Also, the programming languages used (Pascal, PL/I, and FORTRAN, respectively) are unsuitable for teaching large system issues.

8 Conclusions

In introductory computer science courses students are usually assigned programming projects that amount to little more than toy programs. The reason is the students' lack

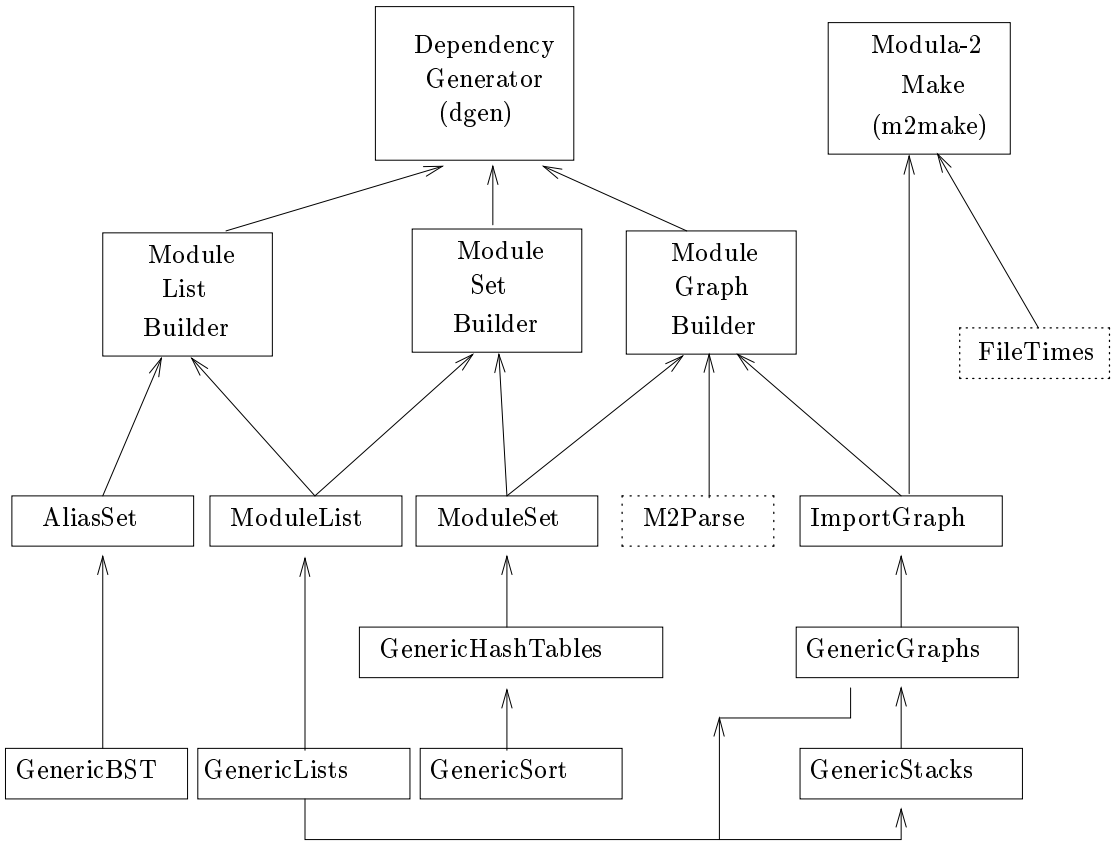

Figure 2: The module structure of the complete project, as envisioned by the instructor. Dashed boxes indicate modules made available to the students. *FileTimes* contains routines operating on file modification times. The stack handler is used in graph traversals.

of programming maturity which makes it impossible for them to successfully manage a complex project from beginning to end. The consequence is that the students will not be confronted with the inherent complexities of large programs until they are approaching the end of their undergraduate computer science program. Even worse, if a programming project course is not required students may not even see, let alone design, a large program until well after graduation. The present course tries to alleviate this problem.

Essential parts of large system design, namely team effort and management, were not a part of the present course. There were three primary reasons for this: Firstly, it was considered important that each student would implement all major data structures and algorithms. Secondly, team coordination takes time and effort which, in this case, could be better spent learning the subject matter. Thirdly, students who drop the class will make problems for the remaining team members. We are not entirely happy with the lack of training in management skills and in future courses this may change.

Modula-2 was chosen both as the implementation- and target language. Modula-2 has many advantages over other languages frequently used in teaching. Other languages, such as Ada or Mesa, which support software engineering principles would have worked equally

well. One important point to consider, however, is that in order to automatically generate dependency graphs, the target language should make module dependencies explicit in the source code. Automatic extraction of dependencies in languages such as C which rely on *include*-files for modularization is difficult and error prone [7]. Such languages should therefore be avoided.

Acknowledgements

We would like to thank Anders Edenbrandt and Sheila Dooley Collberg for their careful reading of this paper. Special thanks, as always, to D. Ed. Nils G. Hult.

References

[1] Jurek Czyzowicz and Michal Iglewski. Implementing generic types in Modula-2. *SIGPLAN Notices*, 20(12):26–32, June 1985.

[2] Stuart I. Feldman. Make - a program for maintaining computer programs. *Software-Practice and Experience*, 9(4):255–265, April 1979.

[3] Will Gillett. The anatomy of a project oriented second course for computer science majors. In *11th SIGCSE Technical Symposium on Computer Science Education*, pages 25–31. ACM, February 1980.

[4] Elliot B. Koffman. The case for Modula-2 in CS1 and CS2. In *19th SIGCSE Technical Symposium on Computer Science Education*, pages 49–53. ACM, February 1988.

[5] Elliot B. Koffman, David Stemple, and Caroline E. Wardle. Recommended curriculum for CS2, 1984. *CACM*, 28(8):815–818, August 1985.

[6] David B. Teague. Computer programming II: A project-oriented course. In *12th SIGCSE Technical Symposium on Computer Science Education*, pages 41–45. ACM, February 1981.

[7] Kim Walden. Automatic generation of Make dependencies. *Software-Practice and Experience*, 14(6):575–585, June 1984.

[8] Laurie Honour Werth. Integrating software engineering into an intermediate programming class. In *19th SIGCSE Technical Symposium on Computer Science Education*, pages 54–58. ACM, February 1988.

[9] Richard S. Wiener and Richard F. Sinovec. Two approaches to implementing generic data structures in Modula-2. *SIGPLAN Notices*, 20(6):56–64, June 1985.