1-1-1986

Producing Software Using Tools in a Workstation Environment

Mark Sherman
Dartmouth College

Robert L. Scot Drysdale
Dartmouth College

PRODUCING SOFTWARE USING TOOLS IN A WORKSTATION ENVIRONMENT

Mark Sherman, Robert L. Drysdale III

Technical Report PCS-TR86-134

(REVISED MARCH 1987)

Producing Software Using Tools in a Workstation Environment

Mark Sherman
Robert L. Drysdale III

Information Technology Center
Carnegie-Mellon University
Pittsburgh, PA 15213

and

Department of Mathematics and Computer Science
Dartmouth College
Hanover, NH 03755

Abstract

We discuss how we taught students to build and use translation, interpretive, editing and monitoring tools in an undergraduate software engineering course. Students used these tools on low-cost workstations (Macintoshes) to build large, group projects. The students' projects used all available features of workstation environments, including graphics, windows, fonts, mice, networks and sound generators. We found that 1) the use of tools increased student productivity, 2) a shift in data structure and algorithm topics is needed to cover material relevant for workstation environments, 3) new topics in system design are required for a workstation environment, 4) traditional material can be easily illustrated with a workstation environment and 5) students enjoyed being able to manipulate the advanced features of workstations in their work, which in turn increased their motivation for and concentration on the course material.

I. Introduction

Dartmouth College has continually reformulated its undergraduate software engineering course to give

students experience with designing advanced systems using modern architectures. Our course is oriented towards the technical aspects of building a large system rather than the managerial aspects, and as such, tries to incorporate the principles and applications of current technologies and trends.

One recent decision was to emphasize the construction of systems using tools in a workstation-oriented environment. Our decision was based on two beliefs about the nature of future software systems. First, construction of large systems will be done by the integration of many pieces, each possibly built in a different way. Many pieces will not be programmed in the traditional sense but will be developed through the use of tools, specified via "very high level languages" or simply taken off the shelf. Second, workstations provide a richer and more sophisticated set of facilities than conventional computers. For example, workstations usually provide substantial graphics facilities (both vector and bit-mapped), network facilities, sound facilities and pointing devices, e.g., a mouse. Although no single facility is unique to workstations, for example, networks existed before widespread deployment of workstations and not all workstations are on a network, the combination of features affects the kinds of decisions that are needed when building a large system.

We believe that we have succeeded in teaching our students how to build and use tools, how to build systems out of pieces developed in heterogeneous ways, and how to exploit the features of a workstation environment. The resulting systems are straightforward to build, easy to use and accomplish their intended functions. Through this article, we wish to share our experience with the novel aspects of our course.

We start with a general discussion of our course. We use section 3 to describe some workstation features that students used. In section 4, we elaborate on some of the tool and system building technologies that

we used. The next section contains a discussion of the program organizations we taught students. In section 6, we briefly list the kinds of computer-user interactions that we covered in class. Section 7 discusses how we shifted our traditional presentation of data structures to accomodate workstation environments. Our experiences with the revised course are given in section 8. We present our conclusions in section 9. The two appendices list the course assignments and the materials used by the students.

II. General Outline of Course

Our course is officially titled *Programming System Design and Development.* We give a series of lectures on system specification and design, project team organization, project construction, and data structure selection and evaluation. Weekly homework assignments illustrate how the principles discussed that week in lecture can be applied in a specific situation. During the latter half of the ten week course, students design and build a project. Our intention is to provide students the opportunity to synthesize and apply the material they learned to a real system. To assist students with selecting a project, we occasionally circulate "requests for proposals" to the general faculty for appropriate projects. Students may select a project from this list or may choose one of their own (subject to our approval on scope and feasibility). About one third of the projects are selected from the returned suggestions. Either the faculty member who made the project request or one of the course instructors serves as the target user who must be satisfied.

Our course is required for all majors in computer science, and a number of nonmajors take the course as well. The prerequisite is a course covering programming and an introduction to computer science. We typically have 25 students per section, most of whom are sophmores. Four sections of the course are

offered each year.

III. Substantial Architectural Features

Students use Apple Macintoshes [Williams83, Apple86] in a variety of configurations in our class. The Macintosh provides many of the sophisticated facilities associated with workstations: graphics, sound synthesizers, networks and a pointing device. We chose the Macintosh over alternatives, such as a Sun workstation, because the Macintosh is relatively inexpensive. In this section, we wish to describe the available facilities to give a feeling of the breadth of possibilities. In later sections, we illustrate how each facility was used as a teaching medium in the course.

The Macintosh provides a bit-mapped display along with a substantial graphics library (Quickdraw). These facilities allowed students to write programs that produced charts, pictures, graphs and other visual results in addition to the typical text and numeric outputs. Further, the conventional graphics features of the Macintosh provide a basis for several other libraries that manipulate higher-level graphics. Such higher-level facilities, such as fonts, windows and menus, provide a basis for sophisticated user interfaces.

A second facility provided by the Macintosh is a sound generator. The generator can be used to produce simple tones or a free-form wave. Like the graphics facilities, the sound generator has an additional layer of software that permits higher-level interactions, such as a speech synthesizer.

Like most workstations, the Macintosh provides network software for both point-to-point and broadcast communication. The machines used by the students in our course were connected together on an AppleTalk network, as were a laser printer and a file (disk) server. Thus students had the opportunity to use

network services, to create new network services and to write distributed applications.

A fifth facility is an analog pointing device, i.e., a mouse. In combination with the graphics facilities, a mouse allows for several kinds of user input that are unavailable with conventional terminals.

The students worked with several implementations of the Macintosh architecture. The Macintoshes provided to the students had differing amounts of memory, secondary storage and screen size. Slightly different releases of software were also used. The Macintoshes produced output for several kinds of printers and could get information from several kinds of pointing devices.

The combination of these facilities provides a large space of alternative methods of computer-user interaction and an immense amount of available software for system development. Therefore the facilities allow a student to experience the need to *engineer* a reasonable system instead of building every piece from scratch. One key to manipulating the vast array of possibilities is the use of tools for specifying and manipulating higher level abstractions that are used by a working system. Without tools, students could only begin to use small subsets of the features. In the next section, we discuss some of the tools used and built by our students.

IV. Use of Tools

Our course placed a heavy emphasis on the use of tools to speed the development process and to aid the maintenance process. We acquired a large set of commercial tools for use by the students, built several more, and taught several techniques for building tools so that students could augment our facilities. Students used two kinds of tools, general-purpose tools and application-dependent tools. The

general-purpose tools include a text editor, compiler (TML Pascal, Lightspeed Pascal), interpreter (MacPascal), linker and low-level debugger. If we had been able to acquire them for the initial version of the course, we would have had several additional system development tools, such as a source-level debugger for the compiled code, a syntax-directed editor and a source control system. Recent offerings used Lightspeed Pascal which provides a source-level debugger and primitive configuration control. Most of the basic system development tools are well known and students had previous exposure to them in an earlier class.

Although programs like compilers really are tools that increase productivity, most students did not view these programs as tools that aided them in their work. Their view was simply that Pascal was the language of the machine; one used the editor/compiler to run one's program. When they wrote a Pascal program, the Pascal code was a running part of the final system. They resisted the idea of building programs that were not part of the final system -- anything a program could do they could do as well, they thought, by hand. To overcome this resistance, we had the students use a variety of application-specific tools, we instructed them on how to build other kinds of tools, and ultimately, we made them build a tool for a homework assignment.

We emphasized four general classes of tools: interpreters, translators, editors and monitors. The taxonomy is our own. One could separate tools into those that are used by a human (e.g., a text editor) versus those that are used by a program (e.g., a window manager). Our taxonomy is intended to describe function rather than method of application, since we feel that an implementation of the same tool can vary from system to system. Although we do not believe that the space of possible programming aids breaks sharply into these four categories, it was a convenient way to discuss and illustrate a variety of tools. Below, we describe each class of tool and give examples of some application-specific tools that we had students

use.

Interpreters

We described an interpreter tool as one that took a specification of an activity and used that specification to perform the desired action. We did not require the specifications to be free-format text, but allowed specifications to be given as templates and encoded descriptions (such as integers). The three most used interpreters were the window manager, the dialog manager and Quickdraw picture interpreter. The window manager [Apple86] can accept the template of a window and perform the necessary manipulations of the template to provide the desired window. The dialog manager [Apple86] provides a stylized way of collecting input from the user. Like the window manager, a template describes how information should be presented to the user and what inputs from the user are acceptable. A Quickdraw picture is a specification that encodes a picture that can be scaled, translated and displayed [Jernigan85]. Naturally, we covered conventional interpreters as well, and students built a simple interpretive calculator as one of their assignments.

Translators

We described a translator as a program that takes a description of an object or activity and converts it into another description of that object or activity. The latter description might be run directly on the machine, it could be given to another translator for further conversion or it could be interpreted.

Students used two translators. One, a program provided by Apple called *RMaker*, translates a textual description of Macintosh resources into a binary form [Hertzfeld86]. Most of the resources are templates to

be used by various interpreters, such as a window template described above. A second system is called *Thunderscan* which is a video digitizer and image manipulator [Thunderware86]. The system can perform two kinds of translation. First, it can take a picture drawn on a piece of paper and translate it (via hardware) into its own special representation that it can later interpret for display. Second, the system can translate from its own representation to a form usable by other Macintosh software, such as the picture interpreter mentioned above.

Editors

We described an editor as a program that manipulates the description of an object or activity without changing the description language. The students were already familiar with a text editor for manipulating their program text. Because the Macintosh has more than text facilities, there naturally exist editors for more than text. Students used four additional editors during the course.

Three related editors were *REdit, ResEdit* and *Dialog Creator.* All three programs are similar in that they allow the programmer to manipulate objects such as windows and dialogs as those objects would appear on the screen. When editing a window description, for example, the window is displayed on the screen. To change where the window should appear, the programmer repositions the window on the screen using the mouse. When the editing is finished, the current state of the window description is stored. The first two editors alter the binary template (Macintosh resource) that can be directly interpreted by the appropriate Macintosh manager. The last editor manipulates the textual template that can be translated by RMaker.

A fourth editor was developed by one of the course staff [Grosz85], and was used to edit simple songs using an interface that resembles a piano keyboard. One could create, edit and save songs built with this

editor, and later have a program play them by giving the saved description to the sound facility in the Macintosh.

Monitors

We described monitors as programs that control or describe the concurrent operation of a system. Many students had experience with debuggers, either the low-level kind used for assembly language programming or with a source-level debugger used in an interpreter, and therefore understood the basic idea of a monitor. Thus students felt comfortable using other monitors that provided information about the correct operation of their system. For example, students who built distributed systems on the network used the network spy program to observe how parts of their system were communicating. Similarly, we had a program that could record and playback user interactions in a system along with a real-time clock. Students used this system to measure the speed of alternative implementations as seen by the end-user.

However, students were less comfortable with the idea of using a monitor tool as a part of their final system. Because most projects could be cast as one large application, students did not view a monitor as a part of the final application. But monitors can be used to coordinate several separate programs into a unified system. To let students experiment with "coordinating" monitors, we gave them two examples: *Switcher* [Towner86] and *Guided Tour Builder* [Seropian85, Clark86].

The switcher program is a poor-man's multitasking system. It allows several programs to be running at the same time in the Macintosh (although only one can use the screen, mouse and keyboard at a time). Thus one can have a word processor, a terminal emulator and a drawing program running at the same time. To use it effectively as a systems building tool, one must be able to write several cooperating programs. One

project group used this tool to integrate graphical input, text input and examination creation. They wanted to build a system that allowed both text and graphics to be attached to examination questions, but they did not want to rebuild complete text and graphics editors. So they built the question editor and, via switcher, combined it with a drawing program and a word processor. Thus they produced a reasonably high quality system with only moderate investments of time.

A second monitor program was the guided tour builder. Guided Tours are collections of programs, documents (files) and a monitor program (called the Tour Guide) that can record and playback sessions of user interactions. The intention behind the software is to provide an inexpensive teaching aid -- one provides a guided tour showing how to use the system. Only one project produced a guided tour of their system, but as mentioned before, other projects used the same piece of software for other purposes, namely recording and analyzing user actions.

V. Use of System Organizations

Because future systems will be built out of small parts, whether hand crafted or generated by tools, a system needs an overall structure for defining what the pieces are and how they should fit together. Because the workstation facility provides so many pieces from which to choose, the need to pick an appropriate system organization is acute. Therefore, we discussed several system organizations that could be used: some conventional, some unconventional.

Many of the conventional organization and refinement techniques that apply to the construction of conventional systems, such as step-wise refinement, top-down design, and layers of abstraction, apply as

well to the construction of systems that use tools in a workstation environment. However, the use of workstation facilities greatly increases the number of examples of standard techniques to which students have direct experience. Consider two examples: data abstraction and layered abstractions.

Students frequently are unconvinced about the utility or application of data abstraction since the usual examples are simple and transparent. Typically, one uses stacks, sets, queues, lists or deques to illustrate an object and operations on those objects. Unfortunately, the notion of information hiding is frequently lost because the same examples are immediately used to illustrate pointers or similar implementation techniques. Further, a list is usually a minor part of an entire system. Thus the students do not really appreciate the value of information hiding, nor how the idea can be applied beyond the textbook data structures. However, most workstation systems provide a large number of data abstractions for use by the students. For example, the Macintosh provides windows, dialogs, fonts, menus and a variety of graphics objects. These objects are more substantial and obviously useful in a system than a stack of integers. Further, most students do not know how the window system is implemented, and so must take the specification at face value. They experience the leap of faith that data abstraction requires. As a result, students appreciate the value of data abstraction and begin to design their systems using the same ideas.

A second example of how workstation facilities illustrate conventional concept concerns layered abstraction. When using languages that do not support Smalltalk-80-like objects, a discussion of layered abstractions is an exercise of unspecified virtual machines and interpreters. However, most workstation environments provide two natural sequences of abstraction layers: graphics and networks. Using the Macintosh as our example, the most abstract facility used by students are dialogs, which are built on windows, which are built on graphics ports, which are built on bitmaps, which are built on bit images. Another example is provided by the layering of network protocols: the unreliable, local datagram is used by

the unreliable internetwork datagram, which is used by the reliable datagram which is used by the reliable byte stream. At each layer, one can clearly identify those details that are being suppressed and those higher-level functions that are being provided. Because students use different layers for different purposes, they get tangible evidence for the usefulness of layered abstractions.

The use of a workstation system allowed us to teach alternative program organizations besides the conventional ones. Two additional topics that we discussed were event-driven programs and window management [Apple86, Rosenthal86]. We feel that these topics need to be understood to develop future systems but are difficult to teach using non-workstation facilities.

When we discuss event-driven programs, we refer to programs that are able to receive any of a number of circumstances and process it accordingly. By contrast, most programs reach a point during their execution where they prompt the user (or file system) for some input and block awaiting an answer. An event-driven program makes no demands on the user -- it waits for some asynchronous action and then processes the event as required. The events could be hardware related: a key on the keyboard being pressed or released, a timer expiring, a packet arriving on the network or a mouse moving. The events could be software related: a window has been uncovered or some task is ready to run. In either case, the event is passed to an appropriate handler for that event and the program as a whole waits for the next event.

The techniques needed for window management are also new to most students, especially where multiple, overlapping windows are the primary output medium for a program. The key difference is one of history. Typical programs on time-sharing systems send a stream of information to a terminal (or file), and once written, never need to reproduce that stream. On the other hand, a window contains a variety of information that represents the state of the program. It is a status display rather than a stream and needs to

be reproduced when needed. There are three common times when the information in a window must be regenerated: when a window is reopened after being closed, when a window is uncovered and when a window is scrolled. When a window is opened, it is usually blank, but many applications allow windows to be temporarily closed and reopened as a way for the user to control screen clutter. The display that used to be in the window must now be regenerated. Similarly when an obscuring window is removed, the underlying window usually needs its contents redrawn. Finally, when a window is scrolled, the contents that were previously hidden must now be drawn. (We assume that the window system does not keep a complete bitmap image of the window's contents.)

To provide the information about how to draw a window, the programmer must keep a data structure with enough information to regenerate the display. This is a foreign concept to many students. They are used to immediate output, even in a graphics system. For example, when their program needs to draw a rectangle, they draw a rectangle in the appropriate window. In a single window (or graphics terminal) system, this approach works fine. But in a workstation environment, the corresponding window could be partially or completely obscured and the need to display the rectangle postponed until far in the future. Therefore students are encourged to separate the manipulation of the program's state (or internal data structures) from the actual drawing on the screen. We teach an approach where programs just manipulate their internal state as necessary, and provide a way to display that internal state at any time. To return to our previous example, drawing a rectangle is a three step process. First one changes the internal data structure to reflect the fact a rectangle has been added to the window. Second, one informs the window system that certain parts of the window no longer show the correct image. Third, on demand, the window is redrawn with the new rectangle. Note that the rectangle might not be drawn right away if the window is covered. Naturally, there are many optimizations that can be made, but such optimizations should be introduced only after the general model is comprehended by the students.

VI. User interaction issues

Our collection of workstation features opens the door for many possible methods of computer-user interactions. Typically, courses in software engineering assume a simple conversational interface with users via a line-at-a-time interface or via a screen-terminal menu system. Because of their graphics, window facilities, dedicated processing, and picking devices, a workstation provides many more modes of interaction. We believe many should be covered in modern software engineering courses so that designers of systems can provide alternatives to users. We cover nine different approaches in our course. We do not make the claim that we are exhaustive, only that we provide a wide exposure to possible techniques.

Our nine approaches are simple line-at-a-time text, forms, button picks, menu selection, command keys, static pictures, animation, sound effects and speech synthesis. Because students are familar with the simple read-a-line, write-a-line types of interactions, we spend little time discussing those in class. The fill-in-the-form approach is familar to the students who used the Macintosh since the commonly-used dialog facility implements the technique directly. Because several tools mentioned above allowed easy creation of forms, students could trivially provide such facilities in their systems. We force the students to experiment with "button picks" by requiring an assignment to use that technique: the available commands for the program are to be displayed in rectangles on the screen and executed when the mouse is pressed in that rectangle. Similarly, we require students to experiment with various kinds of graphical output in some assignments. In assignment four, for example, students have to show the area of the screen they are examining while they perform a search for a particular point. An example done in class demonstrates how animation can be implemented and the value it has in illustrating the changing state of a system

[Glenn85]. Through the use of commercial software, we also illustrate how sound effects can provide qualitative effects [Fenton84, Fenton85] and finally, we show how speech synthesis can be used as an alternative to text for simple messages [Apple85].

A related, but different topic concerns how input information can be obtained. Again, the facilities provided by a workstation environment enlarge the design space of system interactions. We mentioned how a pointing device can be used to pick an object on the screen, but a pointing device can provide other information. For example, one can record the trace followed by a mouse in free-hand drawing, or use derivatives of mouse movement for velocity or acceleration information. One common application of this information is to control the speed of scrolling in a text window.

When we designed the latest revision of our course, we had in mind several other kinds of user interactions, including the use of touch screens, track balls, tablets and head-mounted infra-red tracking devices. Unfortunately, we could not acquire all of the necessary hardware and software in time for use in our course. We do believe that students should be exposed to these technologies so they can better experiment with new kinds of user interfaces.

Because we wanted our students to evaluate alternative user interfaces, we needed some aids to allow students to design alternatives and let users see them. We already mentioned one tool that students applied, the *Guided Tour Builder*. Students were able to record the actions of users working on a certain set of tasks and see how much time was spent trying to reach a goal. A second tool was a screen layout package that allowed students to easily simulate screens of proposed systems without building the interface. A collection of alternatives could be shown to prospective users for their evaluation and feedback. Students used these tools when writing their project specifications. Several students told us

that having this kind of precise image of the system helped them design and build it.

VII. Supporting Data Structure and Algorithms

Students in our course are taught advanced data structures as they apply to building large systems. In the prerequisite course, students learn about stacks, binary trees, sorting and other elementary data structures and algorithms. When shifting to a workstation environment, a subtle shift of the important algorithms and data structures takes place, from special cases to more general cases. For example, one shift was from one dimensional to two dimensional techniques. One dimensional techniques work well if a single datum is presented and processed with a list of other data. For example, in keeping a dictionary of keywords, one takes a single word and compares it against other single words in the dictionary. However, the graphics and window facilities of a workstation use rectangles and points rather than a single datum in a list. For example, to see which point is closest to another point, one needs to search a two dimensional space. Thus we presented the multidimensional version of binary search trees: KD trees (K dimensional). Similarly, we expanded the interpretation of boolean connectives (e.g., and, or, xor) into interpretations on a display, relaxed the assumption of reliable interprocess communication, and showed how iteration can be unrolled onto a collection machines by spliting and joining.

We feel this is an important shift that must take place in courses that teach student useful algorithms and approaches for designing systems in and for a workstation environment. Just as certain topics in operating systems have changed in response to technology (for example, drum scheduling methods are not emphasized but huge memory allocation techniques are discussed), so too the focus and examples used for teaching the underlying algorithms and data structures must include discussions of bitmap models of

computations, operations on a plane instead of a line, and distribution of function.

VIII. Experiences in the course

This course has been taught nearly twenty times over the last six years, though recently we shifted the emphasis of the course to development of systems on workstation environments with tools. Many of the students' experiences have not changed much from our shift: the course still takes a lot of time, the material is quite technical and the pace is very fast. However, we noticed four aspects of the course that convince us we made a good choice in shifting its focus.

First, students claim to have a lot of fun. As Alan Perlis has noted, a prime motivating factor in computer science is to have fun [Perlis77]. Students clearly enjoyed the ability to use sophisticated graphics, sound generators and the mouse. Thus the students concentrated on what they were doing and, we believe, more efficiently learned what we were presenting.

A second phenomenon we observed is that the students' projects were more aggressive or sophisticated. For example, card game programs were always a popular project, but in the past, one person on such a project was assigned the task of writing the software for displaying cards on a simple graphics terminal. After a lot of effort, an arguably recognizable deck of cards was usually implemented. During our latest offering of the course, a student used a digitizer to read a deck of cards for use in their program. In both the previous version and the current version of the course, students had to design an appropriate interface between the abstract cards and the rest of the system. But in the older version, a lot of manpower was wasted on mastering some very low-level command codes that control our graphics terminals. In the current version, the student was easily able to present a realistic visual presentation for the project in a

minimum of time. Thus this student now had additional time to work on other aspects of their system, such as a better algorithm for automated playing. The result: the final system looked better and provided more functionality than equivalent systems in previous years. In general, most projects we saw had more sophisticated user interfaces and performed more complicated tasks.

A third phenomenon was not unexpected: the method and scope of project construction changed substantially. In particular, we saw a marked increase in tool use and generation, and we saw a decline in the total code size of projects.

Since we emphasized the building of systems by reusing existing facilities and applying tools, it came as no surprise that students took this advice to heart. Every tool we were able to find was used by at least one project group. Several project groups built tools of their own. For example, one group built a translator that took a description of a map and countries, and generated the necessary tables for using that world in a strategic game. Another group designed a facility for adding windows of documentation and help to an arbitrary system. A third group built a tool as their project: a sound wave editor. The idea was similar to the music editor we provided, but allowed manipulation and mixing of sounds based on their Fourier representation.

Perhaps a corollary to the increased use of tools and libraries is that projects also required carefully thought-out communication between the pieces. In past years, projects could be patched at the last minute by one part of the system reaching into another part and modifying some data structure that was supposed to be hidden. Since many of the facilities used by students in their projects were truly opaque to the students, the designs of their project had to delineate clearly how various pieces of information were being generated and distributed.

Because of the increased efficiency of system construction, the projects written by the students were substantially smaller than in previous years. A typical project in past offerings of the course required about 200 pages of PL/1 code. Projects using the techniques and facilities in our version of the course required about 50 pages of source code (mostly Pascal, but some of the code was specifications given to tools).

Our fourth observation concerns a phenomenon that did not happen. When we shifted the emphasis of the course, we were concerned that the general algorithms would be too abstract to grasp at the pace we were presenting them. Normally when one shifts from one dimension to n dimensions, one loses a certain fraction of students. However, we thought that the algorithms from computational geometry were crucial for effectively using the newer technologies. We are happy to report that we saw no substanative difference on the students' part in learning the different methods.

IX. Conclusions

We feel that learning how to build systems using tools in an environment that supports good quality graphics, networks, sound and a pointing device is a necessary part of good undergraduate computer science education. We are pleased that we were able to present this material by modifying our current software engineering course. The students who took the revised version of the course enjoyed the material that was covered and were able to apply it when building their own projects.

We believe that the design and offering of a course like ours is a straightforward task, but one that requires a conscious decision on the part of the relevant faculty. We believe that most universities do not have the infrastructure or campus-wide facilites to support a workstation-based project course. Appropriate faculty

members need to specify and acquire new facilities and software for a course like ours to succeed. We do not know of any good substitute: the only effective way for students to appreciate the design and use of workstation environments is to use and build them. Providing a course such as ours requires extensive equipment (we had a 2:1 ratio of students to Macintoshes, 3:20 ratio of Imagewriters to Macintoshes, 1:20 ratio of LaserWriters to Macintoshes, and 1:15 ratio of file servers to Macintoshes), a capable corse staff (we had a 8:1 ratio of students to course assistants) and large library of software (a partial list of our materials is given in appendix II).

X. Acknowledgements

Our course has been evolving to its current state over the last two years, with its final jump onto a workstation (the Macintosh) in the winter 86 offering of the course. People who have taught the course in the recent past have helped in this evolution and include Paul Chew, Tom Kurtz, David Levine and Vivian Sewelson. To make this kind of course work successfully, one needs a large complement of qualified staff. Our able undergraduate course staff, Rob Collins, Larry Gallagher, Jerry Godes, Ed Grosz, Robert Munafo, and John Scott, created a number of the tools, libraries and demonstrations that students used in the course. Funding for the development of the Macintosh version of the course was provided in part by the Interuniversity Consortium for Educational Computing and in part from the Keck Foundation. Apple provided some of our equipment through a grant to Dartmouth College and assisted us in getting several tools they developed. Naturally, the students on whom *we* experimented deserve our thanks for being cheerful participants.

XI. References

[Apple85] Apple Computer Co, Inc., "MacInTalk 1.1: The Macintosh Speech Synthesizer," *May 1985 Software Supplement Update*, (Apple Computer Company, Inc.,) June 13, 1985.

[Apple86] Apple Computer Company, Inc., *Inside Macintosh*, Addison-Wesley, Reading, MA., 1985, 1986, Volumes 1-4.

[Clark86] Cary Clark, *Tour Tools*, Apple Computer Company, Inc., February, 1986.

[Hertzfeld86] Andy Hertzfeld, *Macintosh Resource Compiler*, Version 2.0, (Apple Computer Company, Inc.,) July 10, 1986.

[Fenton84] Jay Fenton, Marc A. Canter, Mark S. Pierce, *MusicWorks*, MacroMind, Inc., Chicago, IL., 1984.

[Fenton85] Jay Fenton, Mark Pierce, Marc Canter, Dan Sadowski, *VideoWorks*, MacroMind, Inc., Chicago, IL., 1985.

[Glenn85] John Glenn, *Binary Trees*, Technical Report DCS-TR86-127, Department of Mathematics and Computer Science, Dartmouth College, Hanover, NH. 03755, September 21, 1985.

[Grosz85] Ed Grosz, *Music Editor*, Technical Report DCS-TR86-114, Department of Mathematics and Computer Science, Dartmouth College, Hanover, NH. 03755, January, 1986.

[Jernigan85] Ginger Jernigan, "Quickdraw's Internal Picture Definition," *Macintosh Technical Note*, Number 21, (Apple Computer Co., Inc.,) Cupertino, CA. April 24, 1985.

[Perlis77] Alan Perlis, "The Keynote Speech," *Perspectives on Computer Science: From the 10th Anniversity Symposium at the Computer Science Department, Carnegie-Mellon University*, Academic Press, 1977, Anita Jones, Editor, p. 1-6.

[Rosenthal86] David S. H. Rosenthal, *Window System Implementations: Denver Usenix Course Notes*, Tutorial Materials, Usenix Technical Conference, Denver, CO. January 15-17, 1986.

[Seropian85] Hasmig Seropian, *A Guide to Making Guided Tours*, (Apple Computer Company,) Inc., March 22, 1985.

[Thunderware86] Thunderware, Inc., Thunderscan Digitizer for the Macintosh, Orinda, CA.

[Towner86] George Towner, "Inside Switcher," *The Software Supplement*, (Apple Computer Company, Inc.,) Vol. I, Issue 3, June 27, 1986.

[Williams84] Gregg Williams, "The Apple Macintosh Computer," *Byte Magazine*, February, 1984, p. 30-53.

Appendix I - Course Assignments

This appendix gives a brief description of each weekly programming assignment. All of our assignments, examinations, handouts, sample solutions, example programs and supporting libraries are available on five double-sided 3 1/2 inch disks (HFS format) from the authors and from the technical report librarian (see Appendix II).

Assignment 1: Learning Pascal

The first assignment introduced the students to the Pascal development system they were using. Students wrote 10 exercises requiring them to read a file, parse a sequence of characters into words, do some array manipulation and perform some simple graphics operations. Each exercise required about 10 lines of Pascal.

Assignment 2: Calculator

The second assignment required the students to implement a desk calculator (infix notation with parentheses and precendence). Students used a line-at-a-time interface provided by the Pascal run-time system to read a line of text, which they parsed and evaluated using a recursive descent design. Students were given routines to convert strings to numbers.

Assignment 3: Linked Lists

The third assignment required the students to build a linked-list package that supported appending a datum to the end of list, inserting a datum in the front of list and deleting a datum from anywhere in the list.

All operations but delete-an-element were done with a line-at-a-time interface. Deletion was done by displaying the list as a series of boxes and lines, and selecting the box to be deleted with the mouse. The changed list was then displayed. Students were given the Pascal code for getting access to the graphics facilities.

Assignment 4: KD-Trees

The fourth assignment required students to create and manipulate a 2-dimensional binary-search tree. A user could place points on the screen using a mouse. The points were added to the 2D tree as they were entered. The user could draw a rectangle with the mouse and the program was to perform a search of the 2D tree to locate the points within the rectangle. The state of the search was to be recorded by drawing lines enclosing the parts of the screen that were being searched. Program commands were denoted by selecting a button with the mouse from a palette on the side of the screen. Students were provided a program outline that provided the palette and command dispatch.

Assignment 5: Window Manipulation

The fifth assignment required students to create a simple window application. The program provided two windows that could be moved, overlapped and resized. Each window could display a collection of rectangles that the user drew. When a user drew a rectangle in a window, the program had to track the mouse and provide visual feedback about the rectangle. The pattern that filled the rectangle could be changed through the use of a dialog box. Desk accessories had to be supported. Menu items for clearing the current window and deleting a rectangle in the window also had to be supported. Students were provided with a skeleton program that provided the all of the standard interface to the operating system

and toolbox, recognized and dispatched all events, and maintained a single window with a gray background.

Assignment 6: Graphs

The sixth assignment required students to use a provided map, place nodes at cities, bridges and other landmarks, connect the landmarks with edges, determine the shortest path between two selected nodes, and display the result by highlighting the shortest route. All interactions with the program were with a mouse. Modes were selected by using a command palette (like in assignment 4); modes were indicated by changing the cursor shape. Students were provided with a skeleton program similar to the one in assignment 4.

Assignment 7: Enhanced Dialog Manager

The seventh assignment required students to build another toolbox manager called the Enhanced Dialog Manager. The new manager would provide easier access dialogs by automatically managing radio buttons and check boxes as well as editable text fields. Students were given several programs that used the new manager for testing.

Assignment 8: Backtracking

Students wrote a Pascal program to generate a solution to the 8-queens problem using recursion and backtracking. No special user interface was specified. This assignment was intended to be simple to allow students time to work on their projects.

Appendix II - Course Materials

We used two kinds of course materials: printed materials such as articles and book, and machine materials, such as programs, libraries and files. We describe each collection below.

II.1 Printed Materials

Most of our printed materials are readily available.

Aho, Hopcroft, and Ullman, *Data Structures and Algorithms*, Addison-Wesley, 1983.

Jon Bentley, "Multidimensional Binary Search Treees Used for Associative Searching", *CACM* , vol. 18, no. 9, Sept. 1975, pp. 509-517.

J. C. Enos and R. L. van Tilburg, "Software Design," *Computer*, Vol. 14, No. 2, February 1981, p. 61-83.

Jerome Friedman and Jon Bentley, "An Algorithm for Finding Best Matches in Logarithmic Expected Time", *ACM Trans. on Mahtematical Software*, Vol. 3, No. 3, Sept. 1977, pp. 209-226.

Leo Guibas and Robert Sedgewick, "A Dichromatic Framework for Balanced Trees", *19th Symp. on the*

Foundations of Computer Science, Oct. 1978, pp. 8-21.

Butler Lampson, "Hints for Computer System Design," *IEEE Software*, January 1984, p. 11-28.

David L. Parnas, "On the Criteria to be in Decomposing Systems into Modules", *CACM*, Vol. 15, No. 12, December 1972, p. 1053-1058.

Robert Sedgewick, *Algorithms*, Addison-Wesley, 1983.

N. Wirth, "Program Development by Stepwise Refinement," *CACM*, Vol. 14, No. 4, April 1971, p. 221-227.

II.2 Machine Materials

Our collection of materials for the Macintosh came from a variety of sources, including commercial software houses, user groups, commercial data services, various computer-network mailing lists, visitors and companies. Many of our sources no longer distribute their materials. Therefore, we have provided a list of equivalent materials by the most common name we could determine, and have indicated where we think one can obtain these materials as of this writing (see key code at the bottom of the list).

Program Name	Description	Where to Get
3-D Edit	3-D object editor	BCS (49)
ADEdit	Alert/Dialog editor	EDUCOMP (222)
Animation Toolkit	Animation frame editor	AAS
Asm	68000 Assembler	Apple (part of MDS)
ASCII chart	(obvious)	BCS (Dev 2)
Atlas	Picture database manager	Kiewit
Base to Base	Hex/Octal/Dec converter	BCS (DA 4)
Bases	Hex/Octal/Dec converter	BCS (Dev 2)
Binary Trees	Binary tree illustrator	Kiewit

Boot Configure	Configuration editor	BCS (Util 3)
ConCode	68000 programming aid	BCS (DA 2)
Copy Disk	4 swap disk copier	Apple
CopyIIMac	General disk copier	Central Point
Coroutine package	Pascal library	Math & CS
Crash Saver	Crash recovery library	BCS (Util 4)
Cursor Designer	Cursor editor	BCS (Dev 3)
DeRsrc	Resource decompiler	BCS (Dev 1)
Dialog Creator	Dialog editor	BCS (24)
Didler	Text file manipulator	BCS (Util 2)
DisAsm	68000 disassembler	BCS (Dev 4)
Disk Info	File manipulation	BCS (DA 1)
DiskUtil	File manipulation	BCS (Util 2)
Dissbits	Graphics package	Kiewit
Drill	CAI interpreter	Kiewit
Dynamo	Animation constructor	BCS (Games 6)
EDialog	Dialog manager replacement	Math & CS
Edit	Text editor	BCS (Util 2)
Event Tutor	Demonstrates event handling	Kiewit
Exception Edit	Speech macro editor	BCS (Dev 2)
Extras	File manipulator	BCS (DA 1)
FEdit	Disk editor	BCS (Util 3)
Fat Bits DA	Mouse tracker	BCS (10)
File Tools	File manipulation	BCS (DA 4)
Font Blaster	Animation-to-font converter	AAS
Font/DA Mover	Resource installer	BCS (DA 1)
Font Display	Font illustrator	BCS (29)
Font Doubler	Font manipulator	BCS (Util 2)
Font Editor	(obvious)	EDUCOMP (42)
Font Librarian	Font manipulation	BCS (Font 3)
Graph Illustrator	Example graph program	Math & CS
Guided Tour Constructor	Demonstration creator	APDA, Mac. Journaling & Guided Tour
Help unit	Library package	Kiewit
Hex Calc	Hexadecimal calculator	BCS (Dev 2)
Icon Editor	(obvious)	EDUCOMP (223)
Icon Maker	Icon editor	BCS (DA 2)
IEdit	Icon editor	BCS (Util 2)
KD Tree Illustrator	Example KD program	Math & CS
KNet	Network stream library	Kiewit
Launch	Program execution tool	BCS (DA 3)
Localizer	Resource editor	BCS (Util 2)
Logic	Example object program	Math & CS
Link	68000 Linker	TML
MacDB	68000 assembly debugger	Apple
MacDraw	Object-oriented graphics editor	Apple
Macintalk	Speech synthesizer	BCS (Dev 2)
Macintosh Toolbox	Libraries	Part of Macintosh Computer
MacPaint	Bit-oriented graphics editor	Apple

MacWrite	Word processor	Apple
Macsbug	68000 assembly debugger	Apple
MacTools	File and disk manipulator	Central Point
Mass Initializer	Disk utility	BCS (Util 2)
Maze	Example network program	Math & CS
Menu Edit	Menu resource editor	BCS (Util 2)
MEdit	Macro text edtior	BCS (Util 6)
Menu Clock	Timer	BCS (Util 3)
MiniFinder	Command shell	APDA (Macintosh System Software for Developers)
Mousometer DA	Mouse measurements	BCS (10)
Multi-scrap	Graphics filer	BCS (DA 1)
Music	Sound editor	Math & CS
New Key Caps	Keyboard mapping tool	BCS (DA 1)
Other...	Desk Accessory test tool	BCS (DA 2)
Overlay	Graphics integrator	Kiewit
Paint Mover	Graphics integrator	BCS (Graphics 2)
Painter's Helper	Object-oriented graphics editors	(BCS 15)
Pascal	Pascal compiler	TML
Peek	Network packet spy	BCS (Dev 1)
Poke	Network packet injector	BCS (Dev 1)
PurgeIcons	Desktop file utility	BCS (Util 6)
RamStart	Memory utility	BCS (Util 1)
RasNix	Command shell	BCS (DA 2)
Read Lisa	Disk utility	BCS (Util 2)
REdit	Resource editor	BCS (Util 1)
ResEdit	Resource editor	BCS (Util 1)
Resume	Crash recovery library	Math & CS
RMaker	Resource compiler	TML
RMover	Resource manipulator	EDUCOMP (222)
Screen Layout	Screen design package	Kiewit
ScrnEdit	Configuration editor	BCS (Dev 1)
Set File	File utility	EDUCOMP (222)
SetFileInfo	File utility	BCS (Util 5)
Skel	Example program	Kiewit
Screen Maker	Graphics translator	BCS (Dev 1)
Skip Finder	Command utility	BCS (DA 1)
Slide Show	Graphics demonstrator	BCS (Graphics 3)
Speech Lab	Speech editor	BCS (Dev 2)
Squeeze File	Text manipulator	BCS (12)
StrCvt	String conversion library	Math & CS
Switcher	Command shell	APDA (Switcher Developer's Kit)
Tasking package	Programming library	Math & CS
Thunderscan	Video digitizer	Thunderware
Uriah Heap	Memory monitor	BCS (10)
Utils	File manipulation	BCS (DA 4)
VideoWorks	Animation editor	Macro Mind
XL/Serve	File (disk) server	InfoSphere

Keys:

APDA (name of the product followed in parentheses):

Apple Programmer's and Developer's Association
290 SW 43rd St.
Renton, WA 98055

AAS (name of program is the name of the product):

Ann Arbor Softworks, Inc.
308 1/2 S. State Street
Ann Arbor, MI 48104

Apple (name of program is name of product, except for debuggers and assembler, which are part of the MDS product):

Local Apple dealer

BCS (the name or number of the disk followed in parentheses):

Boston Computer Society
BCS-Mac 1
Center Plaza
Boston, MA 02108

Central Point (the product is CopyIIMac, which includes MacTools):

Central Point Software, Inc.
9700 SW Capitol Highway, #100
Portland, OR 97219

EDUCOMP (the number of the disk followed in parentheses):

EDUCOMP
2431 Oxford Avenue
Cardiff, CA 92007

InfoSphere (product was XL/Serve, updated now to MacServe):

InfoSphere
4730 SW Macadam Avenue
Portland, OR 97201

Kiewit:

Courseware Development Group
Kiewit Computation Center
Dartmouth College
Hanover, NH 03755

Macro Mind (name of program is the name of the product):

Hayden Software
600 Suffolk St.
Lowell, MA 01854

Math & CS:

Technical Report Librarian
Department of Mathematics and Computer Science
Bradley Hall
Dartmouth College
Hanover, NH 03755

Thunderware (name of the product is Thunderscan):

Thunderware, Inc.
21 Orinda Way
Orinda, CA 94563

TML (name of the product is MacLanguage Series Pascal):

TML Systems
PO Box 361626
Melbourne, FL 32936