Project NORTHSTAR's Software Engineering Strategy for Courseware Development

Anthony V. Edwards

Abstract

As Project NORTHSTAR began developing courseware, it needed to resolve several software engineering issues. In this paper, I will discuss these issues, our solution to them, and our insights gained. The primary users of NORTHSTAR programs are students who infrequently use the same application, so we adopted common user and graphics interface standards to decrease the learning curve when using new courseware applications. Needing a powerful computing environment that would be portable to future systems, we selected the C programming language, the UNIX operating system, and a standard graphics library. Having no dominant graphics standard from which to choose, we created a graphics portability layer that all applications use. With multiple software engineers working together as a team on different applications but all in a common environment, we adopted coding standards to allow code sharing for a higher degree of modularity. Lastly, since we develop courseware applications being written with many possible ideas, our code must be flexible enough to allow features to be enhanced and expanded easily.

General Philosophy

In 1986, Dan Lynch and Mark Franklin planned the direction for Project NORTHSTAR's software engineering strategy. What was written then still holds true today:

> All software development at NORTHSTAR will endeavor to be portable. We simply do not have the staff to develop and maintain a large number of hardware and system specific programs. Thus, we will sacrifice some fanciness and efficiency for the sake of keeping our programs easy to move from system to system. We fully expect that our computing hardware will continue to change at a rapid pace. Our intent is to avoid changing our software at the same pace. There are obvious tradeoffs between portability and things like performance, functionality, and fanciness. We believe we can strike a healthy balance and still maintain a great deal of portability, both into the future and from the past (we hope to also use software already developed elsewhere). We have in mind a "target machine" with certain functionality assumed (powerful CPU, extensive main memory and disk, bitmap graphics with windows, standard language support, and network capabilities). The RT's and other 5M [sic] workstations all share most of this functionality. Even the IBM AT and the Macintosh share some of this functionality. Future trends seem to be heading in our assumed direction. By developing software for this "target machine" we hope to greatly alleviate the problems we may

encounter porting NORTHSTAR software to new systems.[1]

Choosing a hardware and software platform

Project NORTHSTAR started without any equipment or software. The Project's purpose was to integrate computers into the curriculum and produce courseware where needed.

One of the Project's objectives is long software life cycles. To achieve this, the Project a) cannot be restricted to a particular machine and b) cannot be limited by today's technology. The target NORTHSTAR machine was labeled the 6M machine:

1 Megabyte of RAM
1 Megabyte per second network between machines
1 Megaflops (floating point operations per second)
1 Megapenny price range ($10,000)
1 Megapixel graphics display
1 Mouse

Our initial 6M machine was the IBM RT PC:

4 Megabytes of RAM
10 Megabits per second ethernet connection
.4 Megaflops
2.5 Megapennies ($25,000)
1 Megapixel graphics display
1 Mouse

As long as our software targets the 6M machine and does not take advantage of any RT-specific hardware, then our software can be portable to the next generation of IBM computers or other vendor's workstations. At present, we also have IBM 6152s, SUN 3/60s and DEC Decstation 3100s which fit the 6M guideline.

The two operating system choices for the IBM RT were AIX (System V UNIX) and AOS (BSD 4.2 UNIX). We chose AOS because of its popularity among academic institutions.

To decide what programming language the software engineers should use, the characteristics we sought were a) wide portability, b) structured programming, c) data types and pointers, d) easy availability, and e) extensibility of language with user or vendor built libraries. The choice was between Fortran, Pascal, and C. Fortran is widely available, but is not ideal for structured programming and lacked data types and pointers. Although Pascal offers structured programming, data types and pointers, and extensibility of the language, it is not widely portable. C, on the other hand, is widely portable, offers structured programming, contains data types and pointers, and is easily extensible. We chose C because it most closely matched our criteria, particularly the wide portability aspect.

For a graphics kernel, we had the choice of X Windows (version 10), NeWS, GKS, and Andrew (from Carnegie-Mellon University). At the time, none of these standard interfaces were dominant. We could choose a standard and hope it would become the industry standard; we could choose a standard and convert our code to the dominant

---

[1] Lynch, D.R.; Franklin, M.J., December 1986. NORTHSTAR Program Plan, Phase I, Part I, Appendix I: Initial Software Engineering Strategy for NORTHSTAR Developed Programs.

standard when one was established; or we could choose a standard but expect it to change and plan for that change. We chose the latter:

Creating a software portability layer

We want to write software that can be used at other universities. From the start, we have looked for possible problem areas that could prevent our software from running elsewhere. Since most of our software is highly graphical, the graphics interface was firstmost a problem area. Different universities use a variety of graphical libraries: GKS, CMU Tutor, X Windows, NeWS, Andrew, and others. This makes writing programs potentially difficult.

We need portability either a) at the source code level or b) at the graphical library level. Source code level portability means every program has to be written with every graphics standard in mind. When compiling the program, flags must be set to signal the inclusion of the right graphics calls. This is an unacceptable solution because new graphics standards appear or existing ones change which makes source code maintenance very laborious. At the graphical library level, we face the problem of multiple standards.

Our solution is a compromise of these two: we have created our own graphical library that is a portability layer between our applications and the graphics kernel. We call it NGI, the NORTHSTAR Graphics Interface. It contains calls to many graphics primitives for Andrew (BE1), X Windows (V11R3), Macintosh, and NorthTerm (NORTHSTAR's workstation-to-Macintosh interface). Each graphics primitive, such as drawing a polyline, contains code for each graphical library.

Using this portability layer, we only need to write an application once. When we converted from the Andrew graphics library to X Windows, a software engineer worked on adding the new subroutine calls into NGI and then the other software engineers just recompiled and relinked the applications. All of NORTHSTAR's applications were converted within three weeks: two weeks for modifying NGI, one week for compiling/relinking and testing.

By using X Windows, we benefit from its portability. The Project's equipment now additionally includes Sun 3/60s and DEC Decstation 3100s. Since this other equipment supports X Windows, NGI is portable and therefore the applications are portable.

Aside from a portable graphics layer, we developed the need for interprocess/interprogram communication (IPC). We created a layer between the application and UNIX BSD's IPC sockets which are sometimes implemented subtlely different on different implementations of UNIX. If our applications are migrated to a system that handles IPC differently, we only need to modify the library to accommodate the differences.

Using these portability layers protects the portability of our software. Only these layers should need to be changed when porting to a different system.

Making user-interface standards

Once the hardware and system software was in place, the next step was planning and organizing the courseware itself. Our primary audience is students who are only using the system because the courseware is on it. The student may need to use the system once a term, or many times a term. The student may use the system for a given term and then

not use it again until a year later.

For these students, the software must be as user-friendly as possible to make it easy for them to learn rapidly. It is our goal to allow the student to concentrate on his/her homework or research, not on the nuances of the program.

To this end, we created user-interface standards covering mouse use, menu organization, keystroke usage, icon use and organization, and NGI interface features. Once a student becomes familiar with the basic organization and feel of one application, the student can readily familiarize himself/herself with a different one.

For mouse usage, we target a three button mouse. If the mouse has only one or two buttons, then pressing both buttons simultaneously or a combination of a special key and a mouse click are equivalent to a third button. The left mouse button is typically used for selecting, highlighting, or marking a point. The right mouse button is typically used for moving objects around the graphics window. For the most part, the mouse is kept non-modal. However, if different mouse actions are required for different situations, then the mouse cursor must change to an appropriate symbol to signal to the user that the mouse is acting differently. Also, the program must get the mouse out of that mode as soon as any action not related to the mode is attempted (such as selecting a menu option).

In our applications, menus pop-up when the middle button of a three button mouse is pressed or when both buttons of a two button mouse are pressed. For menu organization, there are four basic menu cards: the front menu card, the file card, the display card, and the solve or calc card. The front menu card contains general options for setting up and editing the data the application manipulates. The file menu card contains all the options for recalling and saving the application's data. The display menu card contains options for controlling what is displayed on the screen. The solve or calc card has all the options that control processing or analyzing the data that the user has input. Additional cards may also be added, of course, as needed by the application.

Keystroke usage has been standardized by mapping a set of control keys as fast-keys for menu options. These keystrokes are for options that are always present such as zoom and pan, file recalling and saving, and redrawing.

Icon use and organization has been standardized as well. Through NGI, all icons must appear in an icon bar that is on the left in the window. To encourage the use of common symbols to common meanings, we created an ICON Library that now contains over 40 icons.

NGI's provision of icon bar manipulation routines is just one example of user interface features that may be used by an application that are used exactly the same by all applications. Other user interface utilities include pop-up menus, a scrolltext box, a text line input box, a user-alert-and-confirm box, and a program status box.

To help promote these standards, we have created a model graphical C program, *ngi_model*.c, which contains over 500 lines of C code. It initializes menus and icons; it handles menu options, keystrokes, and icon selections. When a software engineer starts a new graphics application, he/she begins by altering this file. Without any effort, the file already contains code for the standards.

Achieving modularity with coding standards

Modular code consolidates particular functionality into a package that is only accessible through a well-defined interface. Good software is modular. To make good software, you need programmers who write modular code. Unfortunately, it's tough to gauge a programmer to determine how modular his/her code is. So, to ensure a minimum degree of modularity from a programmer, certain standards need be established. We developed this modularity by promoting the development and documentation of libraries and single-purpose programs.

NORTHSTAR has a variety of C routine libraries that are self-contained modules. The NGI, IPC, and ICON libraries have already been mentioned above; the PAR library handles ASCII text parsing, the PCL library handles p-code compilation and execution, the NEP library takes formulas and calculates answers by expression parsing, the GEOM library is a geometry library with line and polygon geometry routines, the UTIL library contains a variety of miscellaneous functions such as stack handling, queue handling, calendar calculations, string manipulation, file name manipulation, and more.

All of these libraries have well-documented routines. To promote up-to-date documentation, we created a manual extraction utility, ***manx***, which creates a manual entry for a particular routine by extracting the routine declaration and surrounding comments and putting them into a particular order for a manual. With well documented libraries, software engineers can effectively share code with each other, thus decreasing development time. One software engineer creates and maintains a library, all others use it. Writing a library is usually overkill for just one application, but saves reinventing the wheel and promotes reliable software. By building on top of existing libraries, better software for less work can be created.

Modularity is also achieved by use of single-purpose programs. Often, when an application performs some non-trivial calculation, analysis of data, or output that is potentially useful to other applications, that section of code is written as a stand-alone, single-purpose program. For example, we have a graphing utility that takes a command file along with numerical data files to plot a variety of graphs; we have a differential equations solver that takes a data file of equations; and we have a mesh generation utility that takes a data file of boundary points. Thus, an application such as an RLC circuit analyzer can generate differential equations to be solved by one single-purpose program and the graphical results are displayed by another.

These single-purpose programs are not limited to being invoked by our software alone. We have been able to alter the Spice application (from University of California at Berkeley) to send output to our own graphing utility which generates higher quality output. Two other programs, one a fluid dynamics simulator and a neural network simulator, have been altered to use our graphics metacommand interpreter thus providing a graphical display.

In addition to being modules that are invoked by another program, each of the modules is a stand-alone program that is available for use by the computing community as utilities. Each of these program modules has a user manual that comprehensively explains all required input data, commands or variations on analysis available, and a full description and explanation of the functionality it provides.

Writing courseware

Having standards, hardware, system software, and modular libraries and programs to use, the system is ready for software development. Developers can range from students to professors to professional software engineers. The cost of developing software increases from student to professor to software engineer. Although less expensive at first, code written by students or professors may have high future costs of maintaining the software compared to code written by a software engineer.

At Project NORTHSTAR, software engineers develop the software and a professor acts as guide, content advisor, tester, and critic for a particular project. This has worked out well; it keeps the expertise in the right place: The software engineer understand the art of programming and the professor understands the subject matter.

Typical interaction between the software engineer and the professor is characterized by a series of short meetings. The first meetings define the application - what functionality will it have, what analysis will it perform, and how it will present the results. The software engineer then develops a draft version that is crude but clearly shows the direction the application is taking. From then on, the meetings between software engineer and professor are iterative - during each meeting, refinements and new features are suggested, and analysis results tested. After a number of meetings (maybe a half dozen or so) of professor feedback, the courseware has a distinct shape, functionality, and usefulness. At this point, the software engineer goes back and makes sure all necessary error checking is being performed, all input is user-friendly, and operation is smooth. While working on an application project, the software engineer must keep in mind flexibility, expandability (both for enhancements and new features), and configurability.

The software engineer must be very careful to write flexible code. The professor may make suggestions that alter the software engineer's perspective on how to manipulate a particular data object. Separating data manipulation from user interface is essential - all data manipulation (creating a data object, setting/changing a property of a data object, displaying the data object, and removing the data object) must be a modular unit. Any facet of the courseware project may be changed because the professor wants it to perform differently; the more modular the code, the more flexible and easier to change it will be.

The software engineer must also plan on aspects of the courseware to be enhanced. The data structure and design of the code must be able to handle these enhancements. An application that plots electro-magnetic fieldlines for point sources, sinks, and dipoles may need enhancement to include uniform fields; the courseware should not need to be rewritten just to add a new data type.

The software engineer needs to plan on new features being added. New data structures should be relatively easy to integrate with existing code; or maybe existing data structures can be expanded. An application that plots electro-magnetic fieldlines should be expandable to plot equipotentials, vectors, and possibly stream flow arrows if they are desired at some point. Many times a professor will be satisfied with some courseware for a particular term. However, when it comes time to teach that course again, the professor will have suggestions for new features, new functionality, or new result presentation. Keeping this in mind ahead of time will save many hours of code rewriting later, whether it's the same software engineer or a different one.

The last point for a software engineer to keep in mind is configurability. If a particular courseware package is being used by more than one professor, other professors may want particular features added or removed from the original design. Instead of having a different application for each professor, a software engineer can have one application that is configurable. Ways of configuring range from command line options to configuration files. Command line options are useful when only a few details need to be configured to enable or disable a feature or trigger a set of changes within the application. For example, a *-short* option may reduce the number of menu options available for a novice user. Configuration files are useful when a large number of features or aspects are changeable or when a particular feature has a range of possible behavior. For example, a configuration may contain such parameters as default settings: nodes on/off, elements on/off, mesh on/off, boundary elements on/off, the user may alter nodes, perform bandwidth reduction after mesh generation, et cetera might be configuration parameters for a finite elements package. The configuration file might be as simple as a list of feature number ids that are enabled or disabled to a file of ASCII commands and parameters.

Having a software engineer write all the courseware is expensive. At NORTHSTAR, our software acquisition priorities follow this order: beg, borrow, buy, and build. If an application already exists elsewhere, we first try to get them to donate it to us (beg). If not, we try to get them to donate it to us with usage restrictions (borrow). Failing that, we consider purchasing it (buy). And lastly, if what we need just isn't available, we develop it in-house (build).

Finishing an application

An application is not finished until the documentation is written. Documentation is vital for effectly teaching new users how the application operates and why it is useful. An undocumented feature is an unused feature. The major sections of Northware (NORTHSTAR software) help files are: purpose, usage, tutorial, command explanation, and reference.

The purpose and usage sections are complementary. The purpose section explains, without making reference to the particular application, what problem exists that needs to be solved; it states the ideal that the application is trying to achieve. The usage section explains how this particular application answers the need; it is an overview of how the application works.

The tutorial section has proven invaluable at NORTHSTAR. This section is non-presumptuous and guides the user through an example session with the application. The tutorial tries to touch upon as many aspects of the application as possible without overwhelming the user and without going into too much detail about the why's and what's. If the user wants to know more, he/she can read the rest of the documentation. At NORTHSTAR, we have found that many users read the tutorial and then immediately start using the application without reading any more. Tutorials typically take 15 to 20 minutes of a user's time. They are typically shorter than 6 pages long.

The command explanation section thoroughly explains all the features of the application: setting up data, altering the data, analyzing the data, and generating output. All pop-up menu options are explained; icon usage is described; and mouse responses are detailed.

The last major section is the reference section. This section ideally prints so it fits on one page. It summarizes all the features by giving one line explanations. Instead of wading through a manual, many users turn to the reference section to figure out how to use a particular feature and then, if necessary, go to the command explanation section for more information.

Finding sources for courseware ideas

A project with hardware, system software, and software engineers is useless without work to do. Finding courseware project ideas that will ultimately be integrated into a class can be non-trivial. We have found that project ideas come from any one of four areas: professor suggestions, software engineer suggestions, pre-existing needs for an application, and re-working existing research software.

Professors are a good source for application ideas; it certainly helps toward integrating the software into a class curriculum. However, a professor may not understand the capabilities and limitations of computers to make a reasonable request. Or, a professor's idea may require more CPU power than is available in a reasonable amount of time. A professor's idea may be easily solved with an existing application, commercial or other. A software engineer and a professor need to meet to discuss what features are desired, what features are reasonable, and how useful the end product will be. From this meeting, the software engineer and professor should agree upon what bare bones minimum is acceptable for the application and prioritize the features; when it comes down to a deadline, the software engineer will know which aspects to work on first.

Staff software engineers are a good source for ideas, particularly if their educational background is similar with the courses being targeted. However, a software engineer's idea is not useful if no professor wants the courseware for his/her class. Courseware without a course is useless. Once a software engineer has found a professor-sponsor, they both need to meet to determine how the idea can be effectly exploited by a class. If, during the development process, the professor's feedback is ignored, the professor may ultimately not use it in his class.

Pre-existing needs are out there, but they need to be found. Software engineers need to talk with professors or graduating students to find out what courses sorely need computer support. Pre-existing needs tend to be better defined than fresh ideas; an application might help students gain insight and solve problems quicker if the computer performed some tedious task that absorbs so much of a students time.

Many professors and graduate students write research software to help them. This software tends have a crude interface, non-flexible code, a tight focus, and buggy. However, it tends to have good analysis algorithms. Putting a research program in the hands of a software engineer - whether it is rewritten or greatly refined - can turn it into a piece of courseware. The software engineer should meet with the research program's author to understand how focused the program is and try to determine how general it can become, what aspects could be simplified, and which components are essential.

Summary

By targeting a machine with particular qualities and writing software in a highly portable and modular fashion on a popular and widely accepted operating environment, NORTHSTAR strives to write applications that will be long lived - surviving ports to different existing machines, surviving ports to future machines, and maintaining educational usefulness.