Porting and Maintaining with X and Motif
A Retrospective View

Paul Davey

User Interface Technologies Ltd.
17-21 Sturton Street
Cambridge
CB1 2SN
England
email: pd@uit.co.uk

ABSTRACT

In 1989 I accepted a job with IXI Ltd. in Cambridge, England, a small software house specializing in the X Window System. The flagship product, X.desktop, was then moving into its third major version (2.0), the first based on Motif.

Initially IXI was establishing X.desktop, and the concept of desktop software for Unix and X, but as time went the initial issues of rapid portability and flexibility, were replaced by other factors as the company, market and product all grew larger and more complex.

Maturing code, increasing numbers of programmers and staff turnover all played their part in making the management of the ongoing project more and more challenging. When overseeing the porting and maintenance of X.desktop in 1992, I found myself dealing concurrently with three major versions (2.0, 3.0 and 3.5), existing on over 30 platforms (with more than 50 active configurations).

After successfully streamlining the porting and maintenance process I was then asked to organize the Motif toolkit team along similar lines in August 1993. This was subtlety different from the X.desktop work since the code originated externally (from the Open Software Foundation), and had to meet open as opposed to proprietary standards.

Keen competition in the market providing Motif development libraries on Sun meant that high quality and ease of use was highly important. As with X.desktop a rapid response to customer problems was another important factor in keeping customers satisfied.

This talk will examine some of the lessons learned about the development of multi-platform applications and toolkits over the software life cycle of X.desktop. Some additional examples are taken from the work porting and enhancing the Motif toolkit.

Although a premier X application X.desktop code was only partly X based, much of it being either Unix system related or just plain data manipulation. The uses (and abuses) of standard Unix software tools such as yacc, lex and SCCS will be illustrated. The parallel development of a problem and bug tracking systems will be described as it resulted in the customized internal tool in use at the end of 1993.

Initial Development

The very first version of X.desktop was a pure Xlib client (Xt was still under basic development at that time), but this only truly existed as a prototype to illustrate the potential of X and Unix with a Macintosh like face. The first version seen by end users (version 1.3) was Athena widget based but retained the ability to build the pure Xlib version for platforms without an Xt and Athena widget port. The cpp conditional macros supporting this remained in the product source code throughout the first Motif toolkit version (2.0).

Lesson one is that your prototypes often end up as the basis for products. Therefore they should be as carefully written and documented as time and circumstance allow. Cpp macros can be used to select different modules. I estimate that 10 to 20% of the X.desktop 2.0 code was un-compiled (and un-compilable), but was too intermixed with the generic and Intrinsics and Motif widget set code to be removed safely in a reasonable time frame.

Designing For Portability

In order to establish X.desktop as the premier desktop for X it had to be easily portable. This was one of IXI's major successes, the quality of the code and its structure being essential to this.

Two major factors contributed to this. Firstly the strength of the X Window System standards, which meant that APIs were clearly defined, particularly so in comparison to Unix. Secondly the use of a virtual interface to the OS where necessary to hide non standard implementations. Known as OSAL (Operating Systems Abstraction Layer) the interface emulated a subset of POSIX, by overloading system or library functions with wrapper functions named transparently by defining and redefining in header files. All usual source code changes (with one exception) were made in just two header files. One setting environment variables for the build processes, the other setting tokens in a per system header file.

This approach worked very well, and so long as programmers were au fait with non standard functions - this knowledge being enhanced by developing in parallel on as many machines as possible standard Unix code (ie somewhere between POSIX, BSD and System V) - porting issues were typically where are the X libraries on this system, or what is the Motif Intrinsics library called today. The most notable failure was to an parallel Unix like machine (which had best remain anonymous), which failed to implement a fork system call properly. This proved somewhat of a handicap on an application designed to launch other tasks.

The nature of X Window System software where applications have binaries and associated resource database files and of a program as configurable as X.desktop (which can be thought of as a graphical shell with more features and predefined functions than say the Korn shell or BASH) meant that after compilation some other configuration had to be met.

With all the main Unix variants then being either AT&T or BSD based this should not have been a problem but at least one port tried to display an icon for the executable file /usr/ucb/man although on that particular platform the /usr/ucb directory did not exist at all and the man command was located in /usr/bin. In retrospect a virtual command layer would have been a boon, a Shell Command Abstraction Layer if you will.

Lesson: Use virtual environments to hide all system differences

Install Scripts

There was also the problem of installing and un-installing the software on client systems. A customer unloaded a tape then ran a script which installed the binaries and other files on the system. This worked without any major incidents, although the un-install script never quite made it to the point of working until a new engineer interpreted the instructions for writing it slightly differently. "Tar the directory onto the tape" was locally interpreted as

```
tar c directory
```

but the new employee gave the commands

```
cd directory
tar c .
```

This did not cause the install procedure to fail, rather than containing all the required files in a subdirectory they were unloaded in the current directory of the customer installing it. The file INSTALL still existed and when run it worked perfectly. Too perfectly since its final line cleaning up was rm -rf *. As Murphy's Law would have it the customer receiving the tape unloaded it the root directory and lost an entire system.

Lesson: Be very wary of wild cards and rm statements in scripts.

Install scripts seem to gain a life of their own. There is always a customer who wants to locate some part of a product a little differently, a previous version of the install script for IXI Motif had become the source of the majority of support calls as it allowed users to place parts of the distribution in half a dozen locations.

Un-installing into a single tree means that it is easy to check for sufficient disc space, and this can be be made to be the installing user's or system administrator's responsibility. If installation fails at the first attempt, or if several related products are to be unloaded, then install trees can be overlaid without problems, and it is easy to delete the software manually.

For convenience files that that might be placed in standard locations (for example /usr/lib or /usr/bin/X11) can be placed elsewhere and then soft-links made manually or automatically from standard locations to files and directories in the application tree.

Even if speed issues mandate some site dependent configuration this should be kept to a minimum and done dynamically at runtime avoiding any dependencies on installed paths. Configuration should be allowed rather than required. Absolute paths are fundamentally inflexible and should be avoided wherever possible. Setting (or deducing) a single environment variable indicating the top of an installed software hierarchy should allow enough flexibility for all file locations. These rules will allow software to be run from any tree, including directly from CDROM.

> Lesson: Keep distributions to a single tree wherever you can and if possible avoid any post install processing.

Using Generated Code

Where software tools are used which produce automatic code the temptation to edit it by hand should be avoided. X.desktop 2.0 had yacc output that was tweaked for efficiency and could not be regenerated automatically. Only a parser rewrite in version 3.0 solved the problem. If generated code must be altered, it should be clearly documented or automated with a tool such as sed or awk.

The use of such tools also requires some care. I have been amazed on many occasions by the lack of knowledge of highly skilled C programmers of the basic variations in user commands and system facilities between BSD and System V based systems.

> Lesson: If you intend to port to these machines and to use the shell and utilities then the same care must be taken as when using system library code. A vendors added value can easily become subtracted value when porting to other platforms.

The COSE initiative and CDE project will surely aid these issues, but due to legacy systems they will not be on every target machine for several years to come.

Version Control

Obviously version control is a must for all serious projects. The choice of systems seems to be threefold in type, viz a standard provided utility, ie normally SCCS, generic public domain utilities (for example RCS or CVS), or complex software engineering tools such as Code Center. While the latter types of tool can be very expensive straight SCCS seems to me to be rather primitive (as it is over 20 years old) and requires a growing number of ever more complex scripts to adapt it to changing requirements.

Other companies or organizations will always use other systems. For instance the X Consortium uses RCS, while system vendors may use their own proprietary tools or plain or enhanced SCCS.

Ideally not a single line of source should ever be duplicated, and functions should be placed in internal libraries so they can more easily be shared between major versions and different products.

In the real world though there is no ideal solution. Projects grow out of early prototypes adapted from other code.

> Lesson: Use the best tools you can get or afford. Plan time for reviewing and if necessary replacing or updating internal tools.

Since programmers like to code there is a danger that code will be duplicated rather than systems reorganized to minimize maintenance. One project I worked on had three separate trees based on the same core code. Needless to say they evolved to be incompatible with each other and bug fixing was scaled up by a factor of three.

> Lesson: Key functions should be placed in internal libraries or modules and extracted into build directories when needed. Time spent extracting generic functionality from project specific modules will be paid back by a factor of 10 or more later in the software life cycle.

Internal Support and Development Tree Hierarchies

Sharing resources such as disc space and CPU time can cause friction when deadlines and tempers are short. Programmers are the worst people to organize their code in large projects. They are often too wrapped up in the small scale details to take larger view. Good internal support saves a great deal of expensive development time and can also provide a project neutral overview of facilities and procedures.

When maintaining large projects a clear network layout will save you a great deal of time. SCCS and source trees are simple enough, but when the source tree contains a distrib tree, (with all the configuration and binary files to be shipped) and that distrib tree is extracted from an SCCS distrib tree at build time then some clarification is needed. For X.desktop we used the following terms for trees:

SCCS - SCCS s. and p. files
OUT - Source code versions extracted from the SCCS tree
BUILD - Trees for building a port in,
These had source files linked into the OUT tree so
that source files could be shared between ports.
SHIP - Tree's containing final distributions, as delivered
to customers.

Tree's named distrib and source were deliberately not used as their names were already associated with the older system.

Procedures were set in place for updating the latest OUT tree only when a fix was stable and the use of multiple OUT tree's allowed a form of product version control. For example, many ports could be built from the 3.0C8-OUT tree, keeping only one copy of the sources on line, reducing not only disc space but also the chances of unintentional changes to the source code.

Automated Building:

Imake, the X Window System meta-makefile tool was quite rough and ready when X.desktop was first conceived. To allow easier porting X.desktop used its own simple set of scripts and regular Makefiles, with (as I remember it) only one problem relating to the inclusion of other Makefiles. Nevertheless as more systems were ported to some extra issues evolved. Most notably amount of manual intervention required became an issue. While it was felt that imake was becoming standard enough to be useful the investment of time to rework the build system to use it was too great. However the shell based system proved to take longer to implement than the developer had predicted, and with hindsight it would have been better to bite the bullet and develop Imake rules for the many tables in X.desktop.

Motif does use the imake system and integrates well with X, however the OSF developers clearly built on machines where either the X Window System was installed in the usual MIT locations (/usr/lib/X11, /usr/bin/X11 and /usr/include/X11), and/or the build trees for X and Motif were in the same hierarchy. This proved to be hard, but not impossible to resolve with the Sun OpenWindows hierarchy under /usr/openwin, and configuration files allowing a (theoretically) hands off build from a makefile now ship with IXI's Motif development kit.

> Lesson: Never trust a developer's schedules. What seems to be a simple task can be found to be more and more complex as work progresses. Using appropriate tools even if you are initially unfamiliar with them can save time at a later stage. It is worth being prepared to support some tools on development platforms lacking them.

Testing:

Testing is of course a vital part of any release cycle, but restrictions of time and the lack of automated test systems for X until recently made X.desktop time consuming to test. What could be done was done. The OSAL implementation had its own tests run as apart of the build procedure, which would diagnose any non-standard system functions and external APIs could be driven by test programs designed to elicit expected responses. GUI based testing was completely manual and was only applied in full to major releases. Basic functionality was tested for each port, initially by the porting engineer, but in time a separate QA department was established. Many problems turned out to be X server dependent but despite good intentions, repeating manual testing 10 or more times with different servers was not practical.

Testing Motif toolkit ports was somewhat more straight forward as the OSF kindly provided a manual test suite at first and later a fully automated verification test suite. The X Test Suite will make a major contribution to improving the implemented standards of the X Window System, making porting engineers' work that little bit easier.

> Lesson: Like the rebuilding of products, software testing is needed repeatedly when working with a large and mature code base. One person's fix is another persons bug. Automated testing provides the best measure of software acceptability.

Enhancements

Many features have been added to X.desktop over its lifespan. Although the original product architect was involved with the majority of these new features, many other personnel carried out design and implementation. For X.desktop as a proprietary product the main issues when adding functionality were, a strong commitment to backwards compatibility, and change control provided by a separate specification. (The issue of how changes were made to that specification is not one that can be covered here.)

For the Motif toolkit enhancements took several forms. Additional demonstration and sample code is a useful, and portable added value but three types of changes were made to the basic Motif source code. These were transparent changes, where a feature might be re-implemented for greater efficiency or for additional fallbacks; translucent changes, which altered the behavior of the standard distribution to match the specification; and added value, where additional features are provided.

Transparent changes have few if any implications for the developer user, but added value needs controlling to avoid accidental use leading to portability problems. As in IXI Motif this can be provided by cpp macros at compilation time, or environment variables at runtime, allowing or disallowing some or all of the extended features. Translucent changes should not need any control, but with development tools used as widely as Motif and X, bugs in the standard distributions become expected behavior, and therefore a de facto standard which should be supported with control as for added value.

Bug Tracking

The original problem tracking system used at IXI was shell script and file system based. Slow, and without file locking, it had many disadvantages but on its positive side side were flexibility and cost effectiveness. In time as the volume of support work rose rapidly, a search was made for a more sophisticated system, but no suitable product could be found available off the shelf and so a back end database system with a customization facility (in Prolog), and a Motif based GUI was purchased.

This meant that the format of the databases making up the system still had to be defined, and much discussion from the support, QA and and development departments produced the first iteration of the actual system design. With a little trial and error a workable system was produced, and after a few months some minor changes were made.

The major design features were that while records are held separately for: problems (reported by customers, and dealt with primarily by the support department); bugs and feature requests, (created by the development team in response to testing or customer problems); and releases (controlled by the QA team); these records sets are all interlinked. Different focuses displayed various subsets of records, with predefined focuses existing for typical use by various teams.

A useful bonus to the universal database structure was that management reports can be generated allowing sensible scheduling of work. In bug fixing team members were encouraged to select their work from the overviews of work outstanding, allowing some personal control over a fairly routine task. The major issues with the system from the users point of view were, the initial learning curve required, the fairly basic GUI (how do you make a database as exciting looking as a desktop full of icons?), and the speed of interactive response.

The IBIS system did prove a valuable resource in planning and distributing work between teams and team members. In my own opinion its largest drawback was the lack of integration into the source code and version control systems. Cutting and pasting SCCS identifiers when you have hundreds of modules and versions is hard to enforce - and of course as a computer scientist I expect the machines to do the tedious work for me.