Putative Software Engineering and the X Window System

Bart Massey
Portland State University
bart@cs.pdx.edu

Abstract

Academic software engineering folks and open source developers are beginning to come to grips with each other's viewpoints on the problem of large-scale high-quality software development. This is leading to tremendous progress in software development techniques and technologies. This paper reviews this trend in the context of the X Window System. The focus is on the standard X distribution: specific areas of interest include revision history and configuration management, defect tracking and quality analysis, and static analysis tools.

1 Introduction

The traditional academic software engineering community is starting to become aware of the growing body of products and practices of open source software development, and the tremendous potential this has to influence software engineering in other domains. At the same time, the open source folks are starting to become aware of some of the difficulties associated with managing large, long-running, loosely-controlled software development projects, and to explore the ways that traditional software engineering practices and tools can help to address the problem. The combination of these trends appears to be highly synergistic, constituting not just a revolution in software, but a revolution in software development.

2 Linux, X, and SE

So far, the principal focus for this convergence has been the Linux kernel. As the most visible symbol of the free and open source software movement, it is an obvious target for attention. It is also a difficult piece of software: large, difficult to maintain and debug because of its hardware interactions, requiring the attention of a goodly number of ultra-experienced developers to understand the subtle interactions involved.

For these reasons, recent years have seen a number of interesting Linux kernel software engineering projects. Lead developer Linus Torvalds has co-developed a modern (though experimental) distributed source management system, BitKeeper [2], for coordinating kernel changes. Torvalds has also constructed a static analysis tool, "sparse" [7], for detecting (sometimes with programmer assistance) a variety of common errors in kernel code. Vendors such as BitKeeper and Coverity [3] have used the Linux kernel source to demonstrate the efficacy of their commercial software engineering projects.

The X desktop shares many of the attributes that have made the Linux kernel a meeting place for software engineers and open source developers. It too is a large body of code: one by many measures substantially larger and more diverse than the Linux kernel. The X desktop also experiences significant software engineering difficulties as the result of its interaction with hardware. The X desktop code is highly visible, perhaps the most visible and enduring part of the UNIX desktop culture.

X has some significant advantages over Linux as a software engineering testbed. A well-written formal protocol specification [8] has been available and frozen over the lifetime of the project. This specification includes an extension mechanism that has been widely utilized; as a cultural norm, these extensions are also well specified. There also exists a high-quality commercially-developed test suite [10] for exercising server-side and client-side functionality.

However, X faces additional software engineering challenges. As one of the older pieces of open source software in continuous use, it faces all the problems of large-scale long-timescale continuous evolution. The requirements for the X desktop are poorly-specified, and change over time in interesting ways. Further, both the client and server sides of the X client-server architecture require engineering effort, effort subjectively of somewhat different type.

The remainder of this paper addresses some of these software engineering challenges in more detail, and sketches the interactions of software engineering with the X Window System.

3 SE and X

A traditional view of the software "life cycle" divides it into roughly 5 phases:

1. **Requirements:** What to build?
2. **Design:** How to build it?
3. **Implementation:** Build it.
4. **Verification and Validation:** Is it built right?
5. **Deployment and Maintenance:** Use it.

For each of these phases, traditional software engineering has specific recommendations that are worth considering in the context of X development. These recommendations are intended to help achieve desirable properties in the resulting system: high quality, high maintainability with low effort, etc.

3.1 Requirements

The decision about what to build is an extremely critical one in commercial development. Building a product that fails to solve the presented problem usually means severe financial loss for the builder. In an open source system like X, the requirements definition is often somewhat more difficult, because the motivations for the work are somewhat more complicated.

For X development, there have been two widely-acknowledged motivators:

1. Desire to provide desktop functionality that is "good"—useful, attractive, valuable, etc.—for users. Includes the desire to be compatible with existing hardware and systems, and the desire to meet user expectations.
2. Desire to be interoperable with existing implementations, and thus to be compatible with existing standards.

Responding to these *user requirements* has been a substantial driver for the development of the X desktop.

Some of the principal misunderstandings by the outside community of the X design and implementation may well be understood as failures of that community to properly analyze the effects of user requirements. For example, there are frequent calls to "remove the network" from X. The usual rationale is that having a networkable protocol "slows X down" or "bloats X" relative to other popular graphical environments. However, measurements of X against proprietary graphical substrates (in the modern era) consistently show comparable speed and size, showing that removing networkability is not necessary to meet the users' speed and . Further, a large subset of the user base uses and demands networkability in X. In this context, it seems clear that requirements-driven development suggests something about the success of X that developers of competitive open-source window systems have failed to understand. The limited success of these alternate projects can be understood as a failure of requirements analysis.

3.2 Design

The traditional way to describe what is to be built is to write a design document. Note that few, if any, of the X components have such a document. This is usually explained by the contention that the design is "documented by the code": oftentimes this contention is borne out by the implementation. Additionally the external server and client APIs, and, more importantly, the X wire protocol are exceptionally well-specified and documented.

There are certain useful ways these design specifications can be used to improve the quality and maintainability of X. For example, it is often useful in large-scale software development to *trace* each specific part of the design to the places in the code where it is implemented. This allows some check on whether important functionality is missing or, often more importantly, whether code has been provided in the implementation that wasn't asked for in the design. This latter is generally considered harmful: "bonus" code is typically not well-designed or well-tested, and is prone to impact the operation of the overall system in negative ways.

3.3 Implementation

The X Window System "sample server" and its associated libraries and applications represent a massive implementation-focused engineering effort. As such, it also represents a significant lock-in: it is difficult to substantially re-engineer a piece of the implementation without starting over. Indeed, Keith Packard's *KDrive* [1] X server represents just such a server-side effort, and Jamey Sharp's and my XCB [6] project represents such an effort on the client side (to discard Xlib: this effort consists both of a redesign and reimplementation). The willingness of the X desktop community to undertake such efforts signals a healthy respect for the implementation.

Most of the implementation difficulties in X are programming-language related. This is not meant *particularly* as a jab at C. Rather, it is more a comment on the state of programming languages: some fundamental problems remain unsolved by current languages (at least, those with reasonably wide followings and production-quality implementations). These problems include lack of sufficiently safe and flexible static type systems, lack of sufficiently safe and flexible modularity mechanisms, difficulty doing cross-language interfacing, platform portability problems, etc. and are discussed extensively in the literature.

These problems are exacerbated in the X environment, where one has to be both a fluent C programmer and a skilled designer in the programming language (a somewhat different skill) to keep subsystems manageable. In an open source setting, where contributions have come from a large number of developers over a long period of time, this is hard to control.

The changes that are starting to take place in X development under the aegis of X.org and freedesktop.org should go a long way toward solving this problem. Increased modularity of subsystems will ease the burden on the C language to handle modularity issues, and will ease the design burden on programmers. Bringing more developers into the mix will help with tackling safety and portability problems in all the ways it normally does in open source projects.

Another technique that has proven important in traditional software engineering but is only now being adopted widely in the open-source community is metrics-based development. It is possible to automatically measure a wide variety of properties of source code automatically, and in the process to learn a lot about the authors, the modules, the schedules, etc., that can be useful in improving software development. My recent preliminary work on obtaining a complete X repository and analyzing changeset information [4] represents one modest step toward the goal of getting good measurements and being able to apply them to improved X development.

There are significant needs appearing for the X desktop. The shift to an OpenGL-based substrate, the increased desire for glitzy (pun-intended) user interfaces, internationalization, operation by physically-challenged users: all of these things will require substantial new design and implementation effort to solve. Hopefully, the X implementation can be brought into line appropriately.

3.4 V&V

There are three legs to the traditional Validation and Verification (V&V) stool: (1) testing, (2) inspection, and (3) formal methods. Of these, testing is the best-known, most widely-practiced, and in some ways the least useful.

The X Test Suite (XTS) is a professionally-designed suite of tests designed to thoroughly exercise the X core protocol, the core X server, and Xlib. Like any test suite, it has twin failings. On one hand, if all the XTS tests pass, it does not assure that there are no bugs. Indeed, one can be quite confident that there are: XTS is a tiny drop in the ocean of possible tests that could be run. On the other hand, if an XTS test fails, it gives no clear indication of the root cause of the failure, thus encouraging inadequate fixes. XTS has an additional idiosyncrasy. It should be extended each time X is: unfortunately, this is not part of the normal development process and never happens. All of that said, XTS is an important part of the X V&V process, and is becoming more so with recent work on it in the context of new X desktop development.

Informal inspection of X Window System code happens quite frequently. Unfortunately, the inspection coverage of X is quite uneven. The recent work by Jamey Sharp on Xlib [9] reflects this. All kinds of interesting discoveries about Xlib were made in the process of understanding its internals—it is likely that much of this code had not been carefully read for years. Techniques exist for much more controlled code inspection that help to produce strong coverage with reasonable effort: unfortunately, they have not been applied overmuch in the open source community. It would be nice to do formal inspections, or at least structured walkthroughs, of selected portions of the X Window System codebase to help improve it; anecdotal evidence suggests that big gains might be present here.

While formal methods are mostly outside the scope of the X work, they have already proven to be useful on at least one occasion [5] and are expected to on others. Some attention should be paid to the possibility of using ultra-lightweight formal methods in the kernel. More importantly, *automated formal methods*, in the form of static defect detection and code analysis tools, is gaining currency rapidly in both the open source and commercial communities. These kinds of automated analyses strongly augment testing and inspection, holding the stool up firmly.

The argument for strong V&V is more than just delivering a better product for developers and users. Recent experience with Xlib supports an important premise of the software quality community. Because higher-quality code is easier to analyze and maintain, it can often be developed more quickly and with less effort than lower-quality code, even taking into account the QA activities needed to produce it. It seems prudent to move X further in this direction.

3.5 Maintenance

The continuous incremental release process used in open source projects such as the X Window System makes it difficult to distinguish between development, perfective maintenance, and corrective maintenance activities. Indeed, the whole range of open source maintenance activities is quite different from those of commercial development.

One point of commonality that they share is the need to properly manage source code. Indeed, this is even more important in an open source project, where code is the principal artifact. X has moved over time from an RCS-controlled to a CVS-controlled environment. In addition, both the Consortium and XFree86.org had humans in the software change loop, closely monitoring and controlling changes to the codebase. With X.org/freedesktop.org moving toward a more open model, the issue of source code management becomes even more crucial. Hopefully the concomitant (enforced) recent move of the Linux kernel to an open source distributed source control system will help to produce a clear direction for the next repository transfer of the X Window System.

Tracking defect reports is another important maintenance activity. The freedesktop.org Bugzilla recently went live: for the first time in its history, the X Window System has an automated defect tracking database. The use of this database for X defect tracking should be encouraged, and efforts should be made to quickly and tightly integrate this with the source management system once one is settled upon.

4 Conclusions

The X Window System represents an amazing piece of software engineering work. Much about how it has been produced and managed has been ideal, and the resulting system has stood the test of time extraordinarily well. X has a positive story to tell the traditional software engineering community.

At the same time, X can still learn a trick or two from traditional software development. By adopting and adapting some standard SE tricks, it may be possible to improve the quality of the X Window System and the productivity of its developers.

References

[1] Eric Anholt. High performance X servers in the Kdrive architecture. In *USENIX Annual Technical Conference, FREENIX Track*, pages 41–50, 2004.

[2] Karl Fogel and Moshe Bar. *Open Source Development with CVS*. Paraglyph Press, third edition, 2003.

[3] Robert Lemos. Security research suggests Linux has fewer flaws. *CNET News*, 13 December 2004.

[4] Bart Massey. Longitudinal analysis of long-timescale open source repository data. In *Proceedings of the First International Workshop on Predictor Models in Software Engineering (PROMISE)*, May 2005. To appear.

[5] Bart Massey and Robert T. Bauer. X meets Z: Verifying correctness in the presence of POSIX threads. In *USENIX Annual Technical Conference, FREENIX Track*, pages 221–234, 2002.

[6] Bart Massey and Jamey Sharp. XCB: An X protocol C Binding. In *Proceedings of the 2001 XFree86 Technical Conference*, Oakland, CA, November 2001. USENIX.

[7] Dave Olien. *Sparse*, July 2003. Presentation slides. URL `http://developer.osdl.org/dmo/sparse/` accessed 13 May 2005 04:31 UTC.

[8] Robert W. Scheifler, James Gettys, Jim Flowers, and David Rosenthal. *X Window System: The Complete Reference to Xlib, X Protocol, ICCCM, and XLFD*. Digital Press, third edition, 1992.

[9] Jamey Sharp. How Xlib is implemented (and what we're doing about it). In *USENIX Annual Technical Conference, FREENIX Track*, pages 51–61, 2004.

[10] Michael Lee Squires and Len Wyatt. An approach to testing X Window System servers at a protocol level, 1987. Available in Xfree86 distribution.