Applying Traditional Unix Tools During Maintenance: An Experience Report

Arthur Tateishi
Ruhtra Consulting Services Inc.
702 - 149 St. George St.
Toronto, ON, Canada M5R 2L9
ruhtra@acm.org

Andrew Walenstein
School of Computer Science
Simon Fraser University
Burnaby, BC, Canada V5A 1S6
walenste@sfu.ca

Abstract

The Unix programming environment provides a rich and fairly standard set of generic tools for the development and maintenance of software. In an age of advanced software tools and environments, it is useful to see how new tools compare against an old baseline. Our experiences from one small exercise in maintenance suggest that simple Unix tools seem to raise the bar of excellence deceptively high.

1 Introduction

This paper reports some of our experiences in planning and performing maintenance tasks on Xfig version 3.2.1, a medium-sized structure diagram editor written in C and X11. This work was performed as part of a comparative demonstration of reverse engineering and maintenance tools. Teams with different toolsets squared off to perform several maintenance-oriented tasks. The teams were all assigned tasks that involved planning a number of changes to Xfig, producing some documentation, and reporting on the program’s qualities. The purpose of this demonstration was to generate realistic experience reports that can serve as grist for the mills of future tools builders.

Our inclusion in this demonstration could be conceived of as a type of baseline comparison or control group. We represented developers using a “standard, vanilla” environment which, in our case, was an ordinary Unix installation. There are many development tools that routinely included in Unix systems [2]. These tools encompass a wide assortment of functionality including editing programs (vi, emacs), finding files (find), searching text (grep), managing compilation dependencies (make), processing text (awk, sed, perl), controlling source changes and configurations (RCS, SCCS, CVS, patch, diff), scripting (csh, sh, tcl, python), and building cross-referencing keyword indexes (tags). The philosophy underlying their design centres on providing powerful independent tools that are composable using pipes and scripts and are loosely and flexibly integrated by simple textual representations.

The computer tools we utilized primarily consisted of vi, emacs, grep, gcc, make, gdb, and man. We made relatively incidental use of a web browser, diff, patch, tar, gzip, and rpm. In addition, we used paper and pen. As far as we know, all of these tools were also available to most of the other teams in the demonstration, however the other teams were instructed to use their own tools preferentially.

Our team consisted of two computing science graduate students, Piotr Kaminski (University of Victoria) and Andrew Walenstein (Simon Fraser University), as well as one professional programmer, Arthur Tateishi (Ruhtra Consulting Services Inc). Before starting we discussed our prior experience with Unix and X11 programming. We all felt comfortable with the level of our Unix and software development experience but only one of us (Arthur) had any significant X11 programming experience. All of us took roughly equal turns at the console while the other two coached or read our written instructions.

This report is structured in a way that is intended to highlight the facts of our experience that are helpful in establishing a baseline. We interpret our present task to include a frank description of our problems and failures but also to include our own interpretation of the tool features we found important (whether they are to us mundane or not). First we will give a brief report on our maintenance activities (we did not keep journals or other records of our activities but we are reporting to the best of our recollections). Next we describe the state of our understanding of the system after performing maintenance and how we arrived at it. Finally we conclude with a discussion of lessons that may be learned from our experience.

2 Activity report

Our exercise involved performing three tasks: (1) producing summary documentation, (2) evaluating program structure, and (3) reconnoitering future maintenance tasks. Although the instructions presented them in that order, when confronted with the first task asking for documentation, we looked at the mass of unfamiliar code before us and
rapidly decided to do something else first. It made sense to us to skip ahead to task 3 and actually try out some of the maintenance tasks first. We later returned to the summary documentation and structure evaluation. These activities are outlined below.

2.1 Maintenance Tasks

We more or less completed the first maintenance task, started the second enough to know how to finish, and rather unsuccessfully finished the third. They took 2 hours, 0.5 hours, and 2.5 hours, respectively.

Task 1: Command panel modification. Most of our time was spent browsing and reading code and in compile-edit-run cycles. We quietly assumed that we would need to insert the appropriate menu item definition in parallel to other X11 structures and commands. We were able to use the filenames to rapidly converge on the correct location (w_cmdpanel.c). We needed to grep through other files to find the appropriate entry points for the new menu items.

Task 2: Adding new arc drawing method. We looked at the filenames in search of something arc related. We found two likely candidates and investigated both. We found the arc drawing functions in d_arc.c and used grep to find where they were called from. This led us to w_modepanel.c. We investigated how to add a new drawing command, and found the mode_switches structures which was easily modified to add a new mode.

Task 3: Bug fix in library object drawing. The first thing we did was verify that the bug was reproducible. We also downloaded the latest version of Xfig from the Internet and confirmed that the bug did not reproduce on the latest patchlevel. Our first inclination was to try a debugger backtrace. This quickly revealed a case of infinite recursion. We needed to recompile the code with debugging symbols to show more information. We tried setting a breakpoint in move_object but could not identify the problem.

We tried to follow the flow of control through the code to understand how compound objects were being repainted. This was difficult. It involved numerous iterations of setting breakpoints and examining program states at various stages. We discovered that a dynamic function callback was involved and tried to find out where it was being set and whether it was being set correctly. This proved fruitless as well. We hypothesized next that the redisplay_region function was buggy, and removed the call to redisplay_curobj and tested it again. This seemed to fixed the problem but we had no confidence that it was a correct solution.

We quickly checked to make sure no errors were introduced in the object’s structure when it was being read from a file. Since the file reading routines were shared with other modules and these were working correctly we abandoned that path of investigation. We noticed that many functions in e_placelib.c seemed to be duplicated in w_cmdpanel.c. We compared the functions with similar names and failed to find any meaningful differences. We decided to trace a move operation on a “normal” (non-library) compound object to find out why it failed to recurse ad infinitum. We discovered that a (global) action flag being reset in this case so we looked at places where the flag was modified. After a bit more browsing, we decided to reset the flag when a library object was being placed on the canvas. This also appeared to fix the problem, but we still did not believe we had a correct solution.

In a quest to discover what was really wrong, we studied the latest Xfig code (i.e., we broke down and cheated). We compiled the code with debugging symbols and placed a breakpoint near the location where the infinite recursion was being initiated. It did not take long thereafter before we realized the source of the bug.

2.2 Documentation and evaluation

Documentation and program evaluation activities consisted of listing directory contents, browsing through source code, drawing on paper, and jotting short answers down onto the experimental handbook. We could not simply “dump out” the structure of the system from our maintenance experience. We made use of wildcard expressions in directory listings to help us look at the named modules. We paid close attention to the #include directives in these modules (this could be acquired by a simple grep ‘#include’ *.ch | less command). Using these resources we began to draw modules with lines connected to them. In this process we made use of the understanding of the basic organization of the program (UI–back-end split, utility modules) that we acquired during the maintenance. It took roughly 20 minutes to get the diagram to the point where we were satisfied that it represented the gross connectivity and organization of the system.

The evaluation portion primarily consisted of a debate about the structure and how easily it would be to maintain the code base. The debate rapidly converged on universal agreement. The questions in the instructions (which one might liken to a checklist for code inspections) helped prompt and organize the debates. Relatively little direct reference was made to the actual code at this point.

— The first problem was that the mode number for placed library objects was in the wrong group and resulted in the wrong branch being taken in redisplay_curobj.
3 Description of knowledge gained

We were instructed to produce system documentation, evaluate the program’s structure, and produce instructions for making modifications. This section reports our main findings.

3.1 System documentation

We discovered that the code exhibited a modularity apparent in the file naming conventions. The breakdown of filename prefixes was as follows:

- d_ drawing operations
- e_ editing operations
- f_ file operations (load/save)
- u_ user interface components
- w_ windowing interface routines

Due to regrettable circumstances we no longer have access to the diagram that we constructed. A short description can be offered however. The diagram we drew was a box-and-arrow drawing of module organization and interconnectivity using no particular architectural formalism. We realized that the code seemed to be structured according to the user interface (UI) decomposition instead of along the (possibly more logical) lines of functionality. The overall diagram had three levels: at the lowest level were common utility modules (such as file operations), the topmost level consisted of UI components, and in the middle were reactive modules that were driven from above by UI callbacks and in turn drove the lower-level modules.

3.2 Evaluation of structure

We were left thinking that the best way of describing the program’s structure is “workable” even though we agreed that the decomposition used would not be our choice if we redesigned it. We were satisfied that the modules were not unnecessarily complicated. Despite an overall endorsement of the structure, we thought of some improvements. For one thing, we felt that the decomposition coupled the structure of the code too tightly with the structure of the interface. We noticed some clones and duplicated code which would suggest possible failures to abstract correctly and which implied future maintenance headaches. We also felt frustrated by the complicated redrawing control. It made use of confusing logic and global variables. It seemed to us that some other design pattern (perhaps a model-view-controller pattern? [1]) would help modularize the state and control. We noted that the change log was so sparse that we could not determine whether the current structure was the original designer’s intent.

Besides these structural problems we also noticed a few other organizational and programming problems. First, there was some naming confusion. Some function names were duplicated in different files and had subtly different code. Furthermore some naming decisions were simply poor—distinct functions sometimes had misleadingly similar names. We also found five GOTOs of which one (in main.c) could be removed simply by placing its code block in a separate function. The other four could be removed with the standard trick of using state variables but it seemed unnecessary to do so since they exhibited a typical pattern of use and their application was deemed judicious.

Despite the above imperfections, we felt that for a medium-sized program these flaws were not fatal. We were quite satisfied that modifying the menu and making a new arc mode was straightforward. We felt the code should not be unreasonably difficult to modify since it is not very large and reasonably well organized.

3.3 Program modification plans

Task 1. To properly add the new menu item one needs to follow the modularization convention assumed by Xfig. This involves moving functions around and creating new files to match the new menus as follows:

Files          Remove w_srcrepl and w_print. Modify w_cmdpanel. Create w_file, w_edit, and w_view.
Functions      Modify init_cmd_panel().
Constants      Modify cmd_switches & cmd_sw_info.

Task 2. To create the new radius-based arc drawing mode ("rarc" mode), we propose copying the circle drawing code as a base and adding another drawing mode to specify a three-point drawing mode. The necessary updates are:

Files          Modify w_modepanel. Create d_rarc.
Functions      Add rarc_drawing_selected().
Constants      Modify mode_switches.

Task 3. To fix to the library object drawing bug, change the value of F_PLACE_LIB_OBJ from 55 to 17 in file mode.h.

4 Lessons to learn

Upon reflecting on our experiences two sorts of issues seem to contain some lessons for future developers of reverse engineering and maintenance tools. The first has to do with how our activities reflect on desirable forms of redocumentation in the context of maintenance, and the second has to do with the combination of comfort and flexibility we found in our familiar tools.
4.1 Maintenance and active documentation

The order and manner that we tackled the tasks seems significant. If at the start we had a set of good and complete documentation (task 1) we might be able to evaluate the structure of the program (task 2) and then efficiently determine how to perform the maintenance tasks (task 3). One might presuppose that the “right” way of tackling these tasks would therefore be to first create good documentation so that the remaining problems would be made easy. However, to put it colloquially, this order might just put the horse before the cart. There is a cost to building good documentation, even with better tools than we had available. Although we were not aware of planning it, what we did seems to make some sense in hindsight. We remember wanting a good structure diagram at the beginning, but to the best of our recollection this desire rapidly abated as we dug into the changes. By actually performing some of the maintenance we quickly and incidentally developed a working understanding of the code of interest. This involvement seemed to contribute to our confidence in our knowledge. For instance, we were not satisfied with our two initial bug “fixes” for maintenance task 3 and this may have something to do with our active debugging of it. In addition the code effectively accumulated our maintenance plans. Also, at the end a diff could have been done to reveal our changes and that would have helped answer the experiment questions.

Our experiences also suggest two other possible reasons for why our particular ordering of tasks was helpful. Firstly, actually performing the maintenance forced us to confront evidence that contra-indicated what we believed. Secondly, the compiler helped us out. It sometimes told us of our mistaken beliefs when we wrote fixes based on them. Although some of this work might be eliminated by good redocumentation tools one wonders if it might just reorganize other parts of it (when misconceptions are fixed, etc.).

Our experience adds evidence to suggestions that maintainers may have only a partial understanding of global structure. After making two changes and debugging another, we still had to refer to the code to document its structure. This fact can be interpreted many ways. It can be considered a positive statement about modularity (that certain maintenance can be successfully performed without system-wide knowledge) or a negative indictment of the state of program documentation or our own (typical?) maintenance processes. The mere fact that some maintenance does not need a global understanding of the code gives us pause to wonder where the cutoff point is in the cost-benefit tradeoff of producing global documentation. Depending upon where this cutoff lies it may be unrealistic to complain that reverse engineering tools are not widely used in routine, medium-scale maintenance work.

4.2 Familiar flexibility

We heavily used the search capabilities of our editors and grep. Although grep searches have some amount of noise due to simplistic pattern matching, we did not find that this noise was problematic. It worked well in combination with less and xterm scrolling, and it integrated well with our shell’s command line history and substitution mechanisms. Without giving it much thought we applied it in many different situations (e.g. redocumentation and call-tracing). In sum, expert use of these flexible tools makes their simplicity very deceiving. Our experiences have been echoed elsewhere [3].

We noticed that the conventions used in the program were designed with Unix style tools in mind. The decomposition and naming of modules were well suited to discovery during simple directory listing. Other adaptations to Unix tools are also popular, such as using naming conventions that are easy to search using grep without extraneous matches and function definition formats that make scrolling to them quick and easy (e.g. with the command /function-name/ in vi).

4.3 Summary

Our team’s experiences in this experiment were effortful and only partially successful. Despite this fact we feel we worked efficiently, flexibly, and comfortably. We think that the efficiency, flexibility, and familiarity of traditional Unix tools makes them a very capable baseline for comparison. These qualities are often hard to achieve. The “standard” tools were created and evolved over decades by expert programmers. Many bad tools have simply vanished. The ones that are left have in many respects kept the standard of excellence high for their successors.

Acknowledgements

We would like to thank Piotr Kaminski for his participation in the demonstration and help in creating this report.

References

[1] E. Gamma, R. Helm, R. Johnson and J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, 1994.