An Empirical Study of Software Porting Obstacles

Dorrit Gordon and Linda Werner

UCSC-CRL-99-07
June 12, 1999

Jack Baskin School of Engineering
University of California, Santa Cruz
Santa Cruz, CA 95064 USA

1. Introduction

This paper presents a brief look at some of the challenges which may be encountered with ported software. Using the experimentation framework, as defined by Basili, Selby and Hutchens [2], to describe my work, the original motivation of my study was to develop an abstract model for the porting process. The purpose of the study was to characterize a porting project with respect to its development lifecycle. The object was the porting process. This study encompassed two domains, the porting team, and the software they were porting. The scope was a single project, and the perspective was that of a graduate student researcher acting as an observer and an intern on the porting team. Using this study definition, the initial plan was to simply participate in and observe the porting process in order to learn enough about it to detail a plan for achieving the goal of the study.

To this end, I worked on an engineering team which was responsible for porting a product written by another company to a new operating environment, and for maintaining earlier versions of the port which were out in the field. My primary job was to attempt to reproduce reported failures, document them, and when possible to find alternate ways to achieve the same goals without producing the failures. When I was unable to find an alternative, another engineer would use my notes to determine the defect and to repair it. Due to constraints imposed by non-disclosure agreements, the companies involved and the specific natures of the products cannot be revealed.

Instead of learning enough to develop a plan to abstract a porting lifecycle model, I learned that this project was an extreme case. As such, it made a poor candidate from which to develop an abstraction intended to guide other projects. It did, however, provide a fruitful ground for studying the kinds of problems a porting team might encounter. This is important knowledge to have in furtherance of the goal of the original study, so I shifted the focus of my study. The new study is motivated by a need to improve the efficiency of porting projects even when the code being ported is seriously defective. The purpose here is to characterize the kinds of problems that inhibit the software porting process and to propose some solutions. Using Basili, Selby, and Hutchens' experimentation framework [2], the object is the porting process. The scope, domains, and perspective remain the same as those of the original study definition.

This paper will present, in general terms, some of the programs worked on, some of the difficulties encountered in the work, and a discussion of how these problems could have been avoided. The goal of this paper is to present some ideas about how porting engineers can minimize the impact of problematic software.

2. The Programs

The software product that was studied consisted of a suite of programs managed through a common interface program. The focus is on one of these programs and the interface.

The program, call it Alpha, is a specialized database program. The database is initially formed by importing a file containing the entries in text form. Once the database is constructed, it can be maintained through the general interface to the suite. In addition to tasks such as adding, deleting, and modifying entries, it is possible to replace the entire database by importing a new file. This database is intended to maintain information used by the other programs of the suite.

The interface is a set of menus which allow the user to configure the interface itself, or to access the interfaces for the individual programs. Interface configuration includes tasks such as specifying the administrator's password, or permitting other users various levels of administrative access to the system. The interfaces to each of the specific programs allow access to whatever features that program provides. For example, the Alpha interface contains the mechanisms for modifying the database.

3. The Problems

This chapter will present three problems I encountered with Alpha: large files could not be imported, changing the administrator's password blocked access to the database, and importing a new file to replace an old one was impossible. The first problem was a reported failure; the other two I discovered in attempting to reproduce that failure.

The report of the import failure stated that importing files failed for large files. I found that the import procedure failed for files with more than approximately nine-thousand entries and an average of about seventeen lines per entry. If there were a maximum supported file size, it should have been noted in the documentation. There was no such notation. Furthermore, nine-thousand seventeen-line entries is not an unreasonable number for a customer to expect Alpha to support. Thus, we can assume that the defect was unknown at the time of release. It requires no unusual tactics to produce the failure; any large file import will do so. Since the defect was overlooked, the stress testing must have been insufficient.

I came across the password problem as I was setting up my test environment. When I initially installed the system I used my own password. I decided to change the password to match the one being used with the other programs under test in the lab. When I made the change, I discovered that I was no longer allowed access to Alpha. Alpha apparently stores the administrator's password separately, although, it would seem reasonable for information shared among various programs in a suite to reside in some shared location. I discovered that if the password was changed first in Alpha, then in the interface, I was able to maintain access. The method I found to change the password was undocumented and did not follow the method originally used to set it. Evidently a significant piece of documentation was omitted, or nobody checked whether the interface and Alpha communicated as expected. Possibly the team which wrote the interface did not know that the password was being used in Alpha or the team which wrote Alpha did not know how to link Alpha's password use with that of the interface.

Once Alpha was installed, configured and running, the testing itself simply involved importing files of increasing size until an import failed. While performing this test I encountered another problem. The first time I imported a file after installing Alpha everything worked fine. Unfortunately, all subsequent import attempts failed to work correctly. The import completed, but the database was not actually accessible. This problem can be avoided by manually performing a number of tasks which are automatically performed the first time a file is imported. Like with the password problem, the documentation for performing this procedure is entirely omitted. It is possible that Alpha was not intended to support importing a file to replace an existing one. In that case, this fact should be clearly stated and the access to the menu system for doing so should be disabled. Better yet, importing the file should be moved to the installation process.

4. How They Got There

All of the above problems demonstrate weaknesses in the system testing of these products. It is evident from the kinds of problems described that the testing was either insufficient or the results were ignored. Obviously, in either case, the resulting software will have defects that should have been caught and fixed (or at least documented) before release.

The first failure, that which precludes the import of large files, shows us that the range of test cases was inadequate to measure the true capabilities of the software. It seems obvious that it is important to test features over a range of input sizes up to the maximum size which the feature is intended to support. If the goal is to support input of any size, then the test cases should at least cover the range which seems likely in the target market. In this case, the file size for which the import failed was well within the range which is required by the target market. If they had tested even one file at a size reasonable to consider as a limiting file size, they would have discovered that the program did not work adequately.

The second problem is an interoperability problem. In the case of the password problem the testing of the interface program was probably done entirely separately from the testing of Alpha. If the feature was tested, the test probably consisted of changing the password and confirming that the interface was then accessible under the new password. Since changing the password is an interface task and not an Alpha task, it was probably never tested with Alpha installed, or, at least, no attempt was made to access Alpha after the password was changed. Again, it seems clear that if programs are intended to work together, the features should be tested in all of the supported configurations of installed programs. There are few enough supported configurations that it is reasonable to perform automated functionality testing for all of them.

The third problem, that of importing files more than once, demonstrates a lack of sufficient test repetition. If importing a file had been tried even twice, it would have been discovered that the feature did not work correctly.

It is possible with both the second and third problems that the software did work correctly. In both cases I was able to find a way to achieve the desired result, although neither method was documented. If these undocumented methods were intended to be used, they should have been documented. In addition to the functional testing, the documentation should be checked for accuracy. In both of these situations, if the type of testing I outlined had been used, with the further provision that the tester follow the procedure described in the documentation to accomplish these tasks, the defects would have been revealed.

5. The Test Suite

The software in question is accompanied by a test suite. According to the license, the ported software may not be released until that test suite is passed. It was easy to see that the test suite was inadequate.

The first clue was when a note at the end of one of the test descriptions said, approximately, "This test might not work. If it doesn't work for you just skip it." If it is permissible to skip tests that fail, then 'passing' the test suite is not especially instructive. It only means that some of the software might have worked in some cases.

In addition to allowing failed tests to be skipped, there was no requirement that the product pass all of the tests simultaneously. This does not guarantee a product that passes all the tests. It is not uncommon for a change in the software to break some feature which had been working. If that feature had already been tested and passed it would not be retested to check that it still worked. An acceptance test suite is only useful if the software must pass all of the tests at the same time. Part of the reason this test suite cannot require retesting the entire system after changes have been made is that the test suite is mostly manual. Even for the smaller programs the tests are cumbersome and can takes weeks to run. With tests this unwieldy it would be unreasonable to try to run them repeatedly.

Reading the test descriptions bore out my hypotheses in chapter 4. There are no requirements that the individual tests be run several times at one sitting. The programs do not have to be tested in combination. Only one value need be used as input to test any particular feature. In general, the tests might be useful to check that a work in progress is on the right track. They are not tests which could reasonably be considered to demonstrate correct and complete function.

6. Dealing With the Problems

In general, the ideal situation would be to avoid porting software which has serious defects. One method of determining, in advance, whether a product is likely to be portable would be to use the Software Engineering Institute's Software Capability Evaluation [1] on the developing company. This evaluation is written to help developers evaluate potential subcontractors and their products. Although not identical, the situation where a company is selecting software to port is similar enough to make this evaluation technique useful. If it is determined that the development method is not sufficiently mature then the quality of the software is suspect and it would be wise to choose software from another company.

Occasionally circumstances are such that a company has no choice but to port a particular product, regardless of its quality. In this case market demand was such that we almost certainly would have needed to attempt to port this software even if we had known how defective it was.

Given that porting engineers are sometimes forced by circumstance to work with seriously defective software, what can they do to minimize the impact the defects have on their work?

I suggest a system of 'pre-testing'. Do not wait until the porting approaches completion to begin testing. Instead test the software on its original development platform. These tests should begin with the test suite provided with the software, if such exists. If desired they can also be supplemented with additional tests that meet the testing standards of the licensee organization.

A method of testing software before the work begins has a number of advantages. First of all, it quickly allows the engineers to develop a thorough familiarity with the software. This should make future work on the products more efficient by reducing time spent learning about it. By starting with tests required by the licensing orangization, the engineers also become familiar, early on, with the tests their product must pass. This should help them plan the work. If the provided tests turn out not to meet the standards of the licensee company, the engineers would have a chance to write and run tests of their own which might reveal problems similar to those found in this study.

There are a number of advantages to finding defects before the port begins in earnest. With luck, the original developer will fix the defects, and return the fixes in time to be included in the initial port. If that happens, the porting engineers have limited their work by avoiding the need to port the defective code, then the fix. If the original developers do not provide fixes in time for the port, at least the porting engineers will not be forced to waste time searching for the root of a defect they have no control over.

In cases where the code is clearly written and where sufficient documentation is available, manual pre-testing is not likely to be worthwhile. Even in these cases, though, automated testing can never hurt. To save time it would be reasonable to run automated tests concurrently with work on the actual port. If the SEI Software Capability Evaluation shows potential for a problematic program, pre-testing to find the exact nature of the defects could save a great deal of time.

7. Conclusion

I observed, and assisted with, the maintenance of a ported software product. The software was badly written, badly documented, and badly tested by the developing organization. As a result, enormous amounts of time were consumed with trying to understand the software well enough to work with it. An organized learning procedure would have reduced this time. Evaluating the licensing company according to the SEI Software Capability Evaluation and running tests of the software on its original platform would, together, have formed a relatively efficient way of learning it. Even when the software does not have the weaknesses exhibited by the software in this study, using these two techniques would bring a level of organization to the learning process. This would be beneficial since it allows the engineer to learn the product efficiently rather than in a haphazard and ad hoc fashion.

References

[1] Rick Barbour. Software capability evaluation version 3.0 implementation guide for supplier selection. Technical Report CMU/SEI-95-TR-012 ESC-TR-95-012, The Software Engineering Institute, April 1996.

[2] Victor R. Basili, Richard W. Selby, and David H. Hutchens. Experimentation in software engineering. *IEEE Transactions on Software Engineering*, SE-12(7), July 1986.