USENIX Association

Proceedings of the 2nd Workshop on Industrial Experiences with Systems Software

Boston, Massachusetts, USA
December 8, 2002

USENIX
THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

Tree Houses and Real Houses: Research and Commercial Software

Susan LoVerso, Margo Seltzer
Sleepycat Software
{sue,margo}@sleepycat.com

Abstract

Sleepycat Software develops and supports the Open Source software product Berkeley DB, the most widely deployed embedded database software in the world. Berkeley DB originated at the University of California, Berkeley, and in this paper, we discuss the differences between research software and a quality commercial product. Over the past years we have acquired an education in configuration, portability, and testing. The key message is that code quality, a willingness to rewrite or discard code when necessary, rigorous adherence to internal standards, and constant policing of ourselves are the key requirements of quality software.

1. Introduction

Many research and commercial software communities struggle with technology transfer, converting a research prototype into a commercial product. While the barriers are occasionally political and cultural they are often technical. Regardless, it is a problem research groups must solve to maximize the impact of their research ideas or to profit from their work.

Building software is analogous to building a house: houses should be designed before construction starts, there are building codes to follow, designs and finished products should be reviewed and formally inspected, and the finished product should be flexible enough to accommodate different uses to which customers might put it.

In this analogy, research prototypes are tree houses. While looking like real houses with many of the fixtures of real houses, they are built for a specific purpose: they do not need to last for decades and they do not need to accommodate a variety of uses. There exists a middle-ground between tree houses and real houses that we will call sheds. Sheds are simple, single-purpose structures that must adhere to local building codes (for example, wiring and location). In the rest of this paper, we will use the analogy between house construction and software creation to explore the process necessary to build quality commercial software.

Sleepycat Software develops and supports Berkeley DB, an Open Source database library. Berkeley DB has a long history, starting in the early 1990s at the University of California, Berkeley, where it began as an Open Source replacement [5] for dbm [1] and ndbm [2]. Olson and Bostic added a Btree implementation and a common index-independent API, and the resulting package, Berkeley DB 1.85, was released as part of the 4.4BSD distribution. Using our analogy, Berkeley DB 1.85 was a rather good shed that became widely adopted. It was relatively well-engineered, had consistent interfaces and a consistent coding style, but was lightly tested and supported only a single class of applications needing non-concurrent, unrecoverable data storage.

Seltzer and Olson developed a prototype of a transactional system based on Berkeley DB 1.85 in 1992 [6]. Using our house analogy, it was a tree house; although it demonstrated that a transactional system could be built on Berkeley DB, the system lacked coherent design, shortcuts were taken to avoid solving hard problems, and it was never stable enough to support mission-critical applications.

In 1996, Sleepycat Software was founded to transform the DB 1.85 shed into a real house. While DB 1.85 was in widespread use, it had many shortcomings. The code reflected the multiple authors that had worked on it and there was little consistency between the different access methods. The package worked correctly on common cases, but boundary conditions could fail. For example, the hash package leaked pages in the database if all the items in a bucket were deleted and that bucket had exceeded a single page. There were no utilities to dump and load data from/to the database. There was no way to permit concurrent updates or even a single updater with multiple readers. Even in the read-only case, each process maintained its own cache of recently-used pages, so the system consumed more memory than was necessary. It was a typical shed: it was just great for storing things, so long as nothing catastrophic happened if the things were lost or even slightly damaged.

Today, Sleepycat Software has ten engineers supporting approximately 200,000 lines of code. Berkeley DB 4.X supports Btree, Queue, Hash and Record-oriented access methods, transactions, high levels of concurrency, recoverability, master-slave replication, distributed transaction management, and a wide variety of APIs (C, C++, Java, Tcl, Perl, Ruby, RPC and many more) on a wide variety of platforms (UNIX, Linux, Windows, VxWorks, QNX and more). As Berkeley DB is a library API, it is used exclusively by software developers, not end-users. Our typical sales opportunity is a software engineer who is building an application with data management needs. As we are an Open Source product, we cannot hide our flaws and the engineers who buy our code know exactly what they are getting.

Sleepycat is a small company with limited resources. Generally one or two engineers have responsibility for a project. Those engineers are responsible for all aspects of the project: initial design, implementation, testing, integration and at least a solid first draft of documentation.

Sleepycat is a distributed company. With no fixed office space and engineers on both coasts of the US and in Australia, we cannot walk down the hall and have a spontaneous brainstorming session or a quick whiteboard talk. Good communication is essential to our success. Similarly, various, rigorous processes are imperative. Due to our distributed nature, we have more process than companies orders of magnitude larger. However process is there for maintaining order, not to get in the way of doing the job at hand. Our process must be different from other team-based reviews, such as the Team Software Process (TSP) [8] because we cannot have review meetings other than those conducted over email or the telephone.

Because mentoring is difficult in a distributed company, Sleepycat hires only experienced engineers. Our engineers have all solved difficult problems. We all know how to read and understand someone else's code, and how to make our own code easily understood by others. We understand that code consistency helps us all. We understand that any untested line of code has a bug in it, by definition, and we use any tools we can find to increase our testing and test code coverage.

This paper describes the practices that have been successful for us. Not all of the practices we follow will scale to a large company. The processes we discuss may work well for small teams. Fundamentally, the one requirement necessary is that management must believe that such processes are important and must make them mandatory and part of the culture. For a large company to succeed with these practices it may be necessary to institute a more formal collection of processes and training such as those of the Software Engineering Institute (SEI) [7]. Even with training, large organizations need to incorporate these processes into the everyday workplace constantly. As an organization grows larger, overcoming an individual engineer's resistance to change becomes a bigger issue.

In the rest of this paper we take you through our process of building software and compare it to the construction process of surveying the land (design), offering building choices (portability and configuration), building to code (coding standards), inspections (testing), warranties (customer support), expansion (rewriting), hitting bedrock (unexpected problems), failing inspections (difficult bugs), and preparing to build the next house (lessons learned).

2. Surveying the Landscape

No one would ever let a builder begin construction of a home based on his promise to, “Build you a great 4-bedroom Colonial,” yet it is common for engineers to begin coding without a detailed and explicit design. Any software engineering book will tell you this is a mistake, and it would be right. However, such books will often also tell you to use a formal specification language and a great deal of rigor to actually design something. We see things slightly differently.

In the construction of a house, the design is comprised of blueprints, plot plans, artistic renderings of the interior and exterior, an electricity plan, a plumbing plan, etc. The goal is that one person could complete the design and an entirely different team could (and usually does) implement it. While the same should hold true in the software world, it rarely does. Designs in our world consist of a few basic components:

- A goal statement.
    - What is the purpose of the design?
    - What are we trying to achieve?
    - Are there different categories of functionality (for example, features the design must provide and features that are nice but not essential)?
- A technical description of the software to be built.This usually contains:
    - a rough code breakdown,
    - a detailed discussion of the effects on the rest of the system and overall integration with the rest of the product,

    - a description of the algorithms (in our case, special attention is paid to concurrency and recoverability).
- A test plan.
    - How will we test this?
    - Can we use or augment existing tests?
    - Do we need a new testing infrastructure?
    - Do we need to build testing hooks into the code?
- Documentation.
    - UNIX-style manual pages.
    - Reference Guide sections. In our product, the programmatic interface is what we sell, and so documentation is absolutely crucial.

Once the design is finished, the fun begins. Designs are distributed to both engineering and marketing, and while one or two engineers will have responsibility for review, comments from everyone are encouraged. Obviously, sending designs to the entire company does not scale, but having the design read by a group of people with different perspectives is key. Typically, we ask the following questions as we review each design:

- Does the design solve the customer's problem?
- Does the interface share the same look and feel as the rest of the system? An interface that is difficult to learn or explain, or inappropriate for a particular language will not be popular with our customers. (For example, the same services are often provided in different ways in different languages such as C, Java and Ruby).
- Does the interface match interfaces in similar products, and will it make sense to engineers experienced in this area?
- Is the test plan sufficient?
- Does the design scale to next year's hardware, thousands of threads and terabytes of data, or are there performance bottlenecks?
- Does the design affect data recoverability?
- Does the design affect existing customer's software and their existing databases? (Upgrading software annoys customers, but they are usually willing to do so. Upgrading databases is a serious problem for many of our customers.)
- How pervasive is this change? Can we introduce this change at this time in our release cycle and still reach stability in time for the next release?

The design phase lasts until everyone's concerns and perspectives have been addressed. Coding will often begin before the design has been finalized, but with the understanding that sunk costs are irrelevant, and it is better to be right than to be done quickly. Consensus is almost always reached, but answers are occasionally dictated by the product architect (and official arbiter of good taste).

When the design phase is complete, we have a workable design and schedule. Nonetheless, the development engineer always needs to change things as the coding proceeds. Just like code is audited before it is committed, design changes are propagated to the review engineers for comment. In general, we have not had big surprises when we follow our design protocol rigorously. However, it has certainly been the case that when we are sloppy during the design phase, we suffer for it later.

3. Offering Building Choices

Just as a treehouse is usually built to fit the contours of the particular tree that is in the yard, research prototypes work only on the specific systems and configurations for which they are built. Sheds are similarly built with the confines of the particular yard in mind. They are small enough to be nimble in their placement. A developer cannot hire an architect to design and build a brand new and different house on every lot on which the builder builds. Therefore, a developer has a selection of house designs, each of which can be altered in limited ways to suit the designs to a homeowner or site, such as reversing or rotating rooms around the core or building the garage at basement level. Commercial software must have the flexibility to be configurable and portable to many requirements and systems.

To the extent possible, projects should choose a development language that minimizes the differences of the underlying environments in which it will run. C or C++ projects can be made portable more easily than Fortran. Java is preferable to C/C++, and a high-level scripting language such as Python or Ruby is the best choice of all. Berkeley DB was forced to use C: it was difficult and time-consuming to make C++ run as quickly as native C and when Berkeley DB was developed there was no C++ standard and available C++ compilers varied in major ways. Also, until quite recently, it was impossible to make Java run as quickly as native C.

Portability code should be isolated to a single area and a single set of files. For example, in Berkeley DB, there are `os` source directories (currently `os`, `os_win32` and `os_vxworks`). Compiler, library and operating system interfaces with portability issues are abstracted to files in the `os` directory. This includes variables such as errno, library interfaces such as malloc, and operating system interfaces such as mmap. The portability layer of Berkeley DB is approximately 3000 lines of C.

There are two advantages in creating a portability layer. First, it is easier to port to new platforms. It is often the case that no member of the development team knows the porting platform (for example, there are literally hundreds of different embedded operating systems, and nobody knows any significant fraction of them). By creating a separate portability layer it is possible for someone intimately familiar with the port platform to port Berkeley DB without having to understand Berkeley DB itself.

Second, software rarely needs the full functionality of more complex system calls such as mmap, and programmers commonly configure such complex interfaces incorrectly. A portability layer allows us to export only the limited, necessary functionality from the system, simplifying the code in Berkeley DB. For example, code to use the UNIX stat system call to determine if a file exists and its optimum block size for I/O is complex, as the stat field which holds I/O block size information is a relatively recent addition to UNIX. Also, the code to use stat to determine if a file exists and is a directory is complex, involving the manipulation of octal byte masks on many historic systems. In the Berkeley DB portability layer we have written that code, but exported it to the Berkeley DB mainstream code as two functions: __os_exists and __os_info, which only return the specific information needed.

For C and C++, language types can also be a problem (for example, using size_t and off_t portably can be difficult). In Berkeley DB, we define internal types of fixed size in our software and abstract them through the portability layer APIs. For example, rather than deal with the fact that an off_t may be 16, 32 or 64-bits on a particular platform, we use "page number" and "page size" variables in our software, both of which are declared to be of type uint32_t to guarantee us 32 unsigned bits, and only convert to the platform-dependent off_t when making the system call in the portability layer.

Portability code should always be selected based on a feature, and never based on a platform. Trying to create a separate portability layer for each supported platform results in a multiple update maintenance nightmare. A "platform" is always selected on at least two axes: the compiler and the library/operating system release. In some cases there are three axes, as when Linux vendors select a C library independently of the operating system release. With M vendors, N compilers and O operating system releases, the number of "platforms" quickly scales out of reach of any but the largest development teams. By using standards such as the ISO/IEC C-language standard[11] (ANSI C) and the ISO/IEC system call API standard[12] (POSIX), the set of features is relatively constrained.

Of course, the default portability code should implement whatever "standard" is most widely available on supported platforms. For C-language applications this will likely be the previously mentioned ANSI C and POSIX standards, at least on platforms shipped in the past decade. The kinds of problems we solve in the portability layer are when systems fail to match the standardized behavior, or differ from the standard for some other reason. For example, the dosFS filesystem distributed with VxWorks does not support the POSIX-mandated semantic that software be able to open a file descriptor, remove the underlying file, and then continue to write to the open file descriptor. We solved this problem by adding a flag to the Berkeley DB structure that contains file handles (currently, either a POSIX file descriptor or a Win32 HANDLE). The flag allows the portability layer code that closes file handles to remove the file after the last close of the file handle. As a result, only the portability layer code needs to handle this problem, and Berkeley DB programmers do not even know that it exists.

Portability choices can be made along either lines of code or compiled files. We have not found any maintainability differences between using compile-time tests to include alternate lines of code, or compile-time tests to use one of a few different files. Generally, we have moved portability code for different platforms into separate files when the implementations diverged significantly (shared memory mapping on UNIX vs. Windows) and left it in a single file when the differences were minimal (using gettimeofday, clock_gettime, ftime or time to find out the current time-of-day). When a platform has separate files, we create a new directory but leave the file name as close to the generic file name as possible. For example, absolute path name resolution for all platforms except VxWorks and Win32 is in os/os_abs.c, while the VxWorks implementation is in os_vxworks/os_vx_abs.c and the Win32 implementation is in os_win32/os_abs.c. The VxWorks filename is different due to a workaround for a Wind River build problem.

Configuration choices should be made at compile-time. A significant advantage in distributing source code is that it allows the package to adapt at compile-time to the environment it finds. This is critical as it allows the package to run on platforms its developers have never seen, and it allows the software's community of users to do their own ports. Only the largest of development teams can afford to buy all of the hardware and hire enough employees to support even a limited number of platforms.

Berkeley DB uses the GNU Autoconf software to do compile-time configuration. Autoconf is a tool for producing shell scripts that automatically configure software source code packages to adapt to many kinds of UNIX-like systems. The configuration scripts produced by Autoconf are independent of Autoconf when they are run, so their users do not need to have Autoconf. While Autoconf scripts are difficult to write and difficult to maintain, Berkeley DB does all its configuration testing for UNIX and UNIX-like systems in around 1300 lines of Autoconf script.

Regardless of your approach to configuration, do not ask the user for system information. The user installing the package does not know the answers to your questions, and this approach is doomed from the start. The software must be able to determine for itself any information that it may need at compile- or install-time.

Finally, portability and configuration issues are greatly diminished by following good software practices in other parts of your development:

- Having a good test suite allows the team to buy inexpensive hardware for testing and then easily run regression tests before releases.
- Distributing the test suite allows the community of users to test their ports before contributing them back to the group.
- Never using the namespace of any other part of the system, (including file names, error return values and function names) ensures that you never collide with another library. Where the namespace is shared, documenting the namespaces you use is only courtesy.
- Encouraging developers to use a wide variety of platforms as their desktop and test machines ensures that the code is always being tested for portability flaws.
- Aggressively keeping the code base clean and the software layers independent makes it less likely that portability problems will slip through.

4. Local Building Codes

As a builder must comply with local zoning and building codes, so must software developers observe company coding standards. When building a tree house, there are rarely zoning laws or permits, and the tree house takes whatever form the builder chooses. Similarly, in a research prototype, the code is in whatever language and form the developer prefers. With a shed, there are some guidelines or local laws. Similarly, when engineers cooperate on a project, they generally work more closely together and use the coding standards, if any, of the larger organization of which they are a part, just as a shed might be painted to match a house. If those standards are inconvenient for some reason, however, they will often be ignored.

When building a real house, there are many participants. The excavator must dig the foundation to match the footprint of the house, the carpenters must build the walls to match the blueprints, etc. There is typically a general contractor overseeing the operation to make sure the subcontractors do their particular jobs correctly and match the specification. Similar processes and policies are necessary for building quality commercial software as well.

The debate over the choice of coding standard could probably go on forever. Regardless, it is simply too difficult and expensive to maintain software built using multiple coding standards, and so some coding standard must be chosen. However, a coding standard is useless without enforcement. Consistency and cleanliness of the code are of the highest priority; the details of the standard itself are a distant second. Not only does Sleepycat have a single standard for coding, we adhere to it religiously. Code audits frequently contain suggestions about clarity, proper spelling and punctuation in comments, points of line-splitting, choice of variable names, and a host of other things that have little to do with the technical correctness of the code, but everything to do with the quality of the overall code base. Newly hired engineers are usually taken aback by the rigor with which we strive for consistency, but over time, they realize that adherence to the local codes is not an option; it's the law.

Code consistency makes it easier to maintain cleanliness in the code. Bug reports and user questions are assigned to engineers not only on a knowledge-area basis, but also on a time-available basis. As a result engineers must quickly be able to understand code with which they are not familiar. Fundamentally, reading another engineer's code is reverse engineering. Consistent use of whitespace and punctuation makes it possible to concentrate on what the code does, rather than how it is formatted. Consistent variable naming (for example, across our code base, a database handle is always a `dbp`) makes it possible to immediately understand the set of operations possible using a particular variable. Not only is consistency and code cleanliness necessary for our daily work, it is necessary for our success as an Open Source company. If a builder built homes that did not meet code or zoning laws, or if the walls were not vertical and straight, that builder's reputation would be tarnished, and business would go elsewhere. The quality of the code matters a lot in an Open Source product. Good code gains you credibility and respect. Putting thought into organization and consistent structure leads people to believe that the same care goes into the code itself, which we believe to be true in our case.

Conversely, sloppy code, or code that produces warning messages when it is compiled, or spelling errors in README files or error messages, leads customers to believe the engineering behind it is also haphazard and sloppy. We regularly have customers comment on the quality of our code: "I must say, your code is excellent," or, "It is a real pleasure to read." That feedback not only strokes engineering egos, it is every bit as important to a software development organization as it is for a builder to hear, "Your workmanship is excellent."

Maintaining consistency at Sleepycat is easier than it is at most organizations due to our size and our having an experienced and disciplined engineering team. However, we have processes in place to catch mistakes: all code changes are audited, and audits include checks for stylistic violations. Before a release cycle, we make a pass over the code looking specifically for coding standard and naming mistakes and spelling errors in quoted strings. Compilers are run with "additional warning" flags and any warnings found are cleaned up. We regularly run code-cleaning tools such as lint and others. All engineers are expected to fix violations they find when reading code, regardless of why they were reviewing the code in the first place.

5. The Home Inspection

Typically, completion of a tree house consists of the builders declaring, "It's done!" and 30 seconds later, the children are using it. When a shed is completed, there might be a quick inspection by the local housing authority, and the homeowners can then move their lawn mower inside. When a house is completed the process is more involved. With a reputable builder, the process is verified on a recurring basis so there are not any surprises when the building inspector checks the finished product. After receiving certificates of occupancy from the local authorities, the builder does a formal walk-through of the residence with the new homeowner. New owners perform unit testing on the house: turning on lights and water, flushing the plumbing, and so on. It is the same way with commercial software. Research prototypes rarely undergo rigorous testing, and need only work in a limited setting. Larger software projects might undergo some testing, but are still often expected to be extensively fixed after release. A commercial software product may not survive if it is filled with severe flaws upon release. Maintaining buggy, released software costs the industry billions of dollars every year [4], so every bug found in testing translates into long-term savings.

Given our small support organization, the high level of algorithmic complexity in our code, and the demands of our customer base to never lose data, we live and die by testing—we simply cannot afford to debug problems in the field and must find them before release. We currently have over 40,000 lines of Tcl, testing Berkeley DB's functionality. Of course, we run code coverage tools over test suite runs so we know what areas of the code are not getting covered by our tests. Currently our Tcl test suite covers about 70 percent of our code base by lines of code (a good level, given the difficulty of exercising error code paths in the typical application). Our test suite is composed of hierarchical layers:

- At the lowest layer we have tests of a particular function in the system (for example, does the system handle items that exceed the page size).
- At the next layer, we wrap the lowest layer tests in drivers that run each access method through every test with a variety of parameters (for example, page size, encryption on or off, and so on).
- At the next layer, we test different system configuration options: transactions, locking, replication, RPC. For each configuration, we run all tests in the lower layers.
- At the next layer, we invoke the different tests through each of the high-level APIs (C, C++ and Java).

Not surprisingly, the test suite takes days to run. It is modular enough that engineers can run pieces of it to test code changes, but producing a release implies running the full test suite on a wide variety of compiler and operating system combinations. However, every bug we find in testing is a bug customers do not find, and in a system as complex as ours, many of the bugs we find simply could never be debugged in the field.

However, our test suite, written in Tcl, has limitations. Standard Tcl is not multi-threaded and therefore our test suite is serial as well. In addition, scripting languages are still too slow to stress the library's performance. For these reasons, we have two large application server programs (intended to mimic customer application behavior) that we run continuously on high-end test hardware. The layers of unit testing described above test the library's functionality. The test applications test the library's performance and concurrency control, as well as resistance to non-deterministic behavior, such as random system failure. In an earlier release, our most serious mistake was failing to run our test applications during the development cycle, resulting in an unexpected slip in the release. Tens of thousands of successful iterations of our test applications is what gives us the confidence that a new release is ready for real-world workloads.

6. Warranties and the New Homeowner

When a general contractor builds a house, and the new owners move in, there is a period of time during which the builder will come back and fix anything that is not working or is not quite right. In our business, this is called customer support. Sleepycat has a support organization to track requests and direct them to the appropriate people. However, once the request has been handed off to an engineer, the customer and engineer are in direct contact. This is for several reasons:

- Potential customers are likely evaluating our competitors' products while waiting for us to respond. We need to respond quickly and correctly and not let support requests languish for too long.
- Engineers often select products based on the quality of the support. If they believe the vendor organization will support them as they learn a new API, they have a strong incentive to buy from that vendor.
- We are selling engineering software to smart people. It would be difficult to train support people that could handle a significant fraction of our support questions without being engineers.
- It is important that engineers understand who pays their salary. Contact with customers is an important source of feedback both on future features for the product as well as how the software is perceived, that is, how well the engineers are doing their job.

Sleepycat uses all of its engineers (including our chief architect) in support tasks. For this reason the quality of our support eclipses our competitors, and no single engineer is tasked with the boring work of support. Engineers cannot close support requests; the close must be done by the support manager, who has responsibility for tracking requests and ensuring that the customer is happy at the end of the day. Our engineers behave like a professional support organization with the directives:

- Be polite (no matter what).
- Respond as quickly as possible; support is more important than new code.
- Answer the question as completely as possible.
- Never send out an untested patch to a supported customer.
- Include a link to the appropriate web page if you give them a documentation reference.
- Be grammatical, spell-check all email, and format it so it is easy to read and easy to reply. Customers assume that sloppy email reflects sloppy code.

Sleepycat also treats requests from unsupported users similarly to requests from supported customers, although at a somewhat lower priority. Often, unsupported users find bugs and have good ideas for features. We also believe that helping academic and other unsupported users become comfortable with our product and API will result in future commercial sales.

Sleepycat maintains a complete log of all interaction with customers on every support request. We have had cases of customers wondering what had happened with their support request and our log has been useful in helping them identify errant email forwarding and other black holes into which email fell. In general, our guiding principle is that customers desperately want to talk to a technically competent human if they are reporting and tracking a real bug, and it is our job to provide that human being and a resolution as quickly as is possible.

7. Starting to Dig the Expansion

After you have lived in a house for a while, sometimes your needs change. The family size grows, new room uses are sought. Initially, for a growing family, one may take the faster solution of, say, finishing part of the basement and turning it into a playroom. However, depending on the growth, ages and needs of a family, more drastic measures may be needed. Aging grandparents may be moving in and their needs are different from a toddler's. It might be time to reconfigure some of the existing rooms and add an addition for the changing needs. So it is with software and rewriting particular subsystems.

There has been much discussion in the community about the pitfall of rewriting entire systems from scratch [9], however there has been little discussion concerning the trade-off between rewriting and maintaining subsystems that were written under different conditions than currently exist. As one maintains a product over time, one discovers parts of the code that have become complex, brittle, and/or simply ugly. The code did not start out that way, but as features were added, fundamental assumptions upon which the code is based ceased to be valid. For example, when we built the portion of our system that handled the recovery of file creation, the system observed the rule that every file contained one and only one database. Sometime later, we added subdatabase functionality that let us store multiple databases in a single file. Suddenly all the assumptions we made about the state of a file during an open were no longer valid. Then we added replication to our system which further changed the rules about opening and creating files. With much painful debugging, we were able to make the code pass our tests, but the result was a largely unmaintainable codebase. We had a storm brewing in the code; nobody admitted to really understanding the code, and engineers were loathe to touch it for fear of breaking what seemed to work.

It was time to rethink our entire approach and rewrite this part of the system based upon the new state of the world. Since we were doing that, we had the luxury of considering new functionality as well, and we decided that we could extend our transactional support to provide the ability to group multiple file system level operations in a single transaction (e.g., begin, create file foo; delete file bar). It is terrifying to contemplate rewriting parts of the system that work, but even more terrifying to contemplate leaving fragile code in your product. The key enabler allowing us to rewrite this central component of our software was our test suite and testing infrastructure. Without the test suite, we could never have considered this rewrite. We began the rewrite with a new set of assumptions and designed it based upon a simple set of four file system primitives dealing with modification:

- create files
- remove files
- rename files
- write to files

The design phase lasted about a month with a single engineer owning the design and the rest of the team commenting on it and pointing out the problematic cases. Each engineer brought their own set of biases, assumptions, and knowledge about particular subsystems to the table and the end result was that we debugged significant parts of the design before the first piece of code was written.

The rewrite took about one staff-month of actual coding, one staff-month of unit testing, and a final staff-month of new testing, that is finding and fixing bugs that were discovered as the test suite was enhanced to cover the new functionality that was added. The end result of this rewrite was increased functionality and cleaner code that we no longer fear touching. It was neither easy, painless, nor fun, but it was important and the end result is enormously satisfying.

Perhaps the most educational aspect is the constant tension between doing it right from start to finish and having to rewrite parts of the new code as you go. As the development progressed, there were phases where certain functionality was working, but we would discover a problem that required changing the working functionality. The engineer on the project always resisted such changes, and it took strong and persistent management pressure to ensure the end result was not as brittle and complex as the original code. In general, engineers resist changing things they have written and debugged with good reason; however, sunk costs are irrelevant if the path you are on is headed for disaster. It is absolutely essential to be willing to backtrack. The engineer involved may not realize when it is time to backtrack, which is why someone else must be close enough to identify the need.

8. Hit Bedrock, Stop Digging

Suppose you decide to add an in-ground pool in your backyard. Someone from the pool company comes to your house and surveys the yard, determining where various obstacles might be. Everyone agrees on the pool design and area and the price to install it. Work merrily commences. However, after digging down several feet, the excavator hits bedrock. Work ceases and the options must be considered. Should the bedrock be drilled and blasted (with the unlimited expense paid for by the homeowner) or should the pool be moved, or should the pool just be shallower than originally planned?

Sometimes design flaws are only discovered during implementation. It is important to be willing to admit, at any time during the life of a project, that the wrong path has been chosen and that it is time to step back and rethink the entire design. The brute force alternative, of hitting the round peg really, really hard until it goes in the square hole, will be more expensive in the long term.

Sleepycat recently had one such experience when adding checksum and encryption support to our database product. All I/O in the product is done at a page-level granularity, so we chose to checksum and decrypt/encrypt each page as it was read/written. Cryptography support requires we store 36 bytes of crypto initialization vectors (IV) and checksum data on each page of the database.

The design had to satisfy two additional criteria:

- We did not want to use 36 bytes of space per page unless the user wanted checksums and encryption, as 36 unused bytes on a 512-byte page is a significant overhead.
- We did not want to change the physical page layout written to disk because customers using this release would then have to upgrade their databases (even if they did not need the checksum and encryption functionality).

A Berkeley DB database page begins with a collection of header information (25 bytes), and then has an array of page indices referencing data items on the page, growing forward. The actual page data begins at the end of the page and grows backward. Much of the Berkeley DB software manipulates page structures: getting a reference to bytes at a specific index, computing the first free byte on the page, computing the last free byte on the page, computing the remaining free space, and so on.

In the original design, the engineer decided to retain the original page header. The idea was to put the checksum and crypto information at the end of the page. That would mean only the free space calculation code would need to be adjusted, and code dealing with indices and other parts of the page would remain unchanged. The design did not modify the page header at all, so no upgrade would be necessary. This design was simple, clean, and passed review. Implementation was conceptually simple, as the bulk of the changes were in one header file and one page-related file. After implementation, initial testing of both checksum/crypto pages and standard pages passed. The code was committed into the main source tree.

However, when full test suite runs were made, things started breaking. The problem was that the page size was used for many calculations that were not immediately obvious. Also, there was an assumption in the code that the end of the page is the end of the page's data space. The immediate reaction was to begin bug-fixing and bandaging the code, and adjusting the computations using the page size. However, after some days of this effort, the engineering group remembered the first rule of holes: "If you find yourself in one, stop digging." It was time to climb out of the hole, revisit the design, and change direction.

The alternative solution was to place the checksum and IV at the end of the page header but before the array of indices. The coding changes were larger than the original solution as the location of the database page indices was no longer fixed. However, the necessary code changes were largely flagged by the compiler, and the solution no longer broke in subtle ways: it either worked or broke dramatically.

Coding the original paging scheme, testing it, attempting to fix it, discarding that change, and then repeating the procedure with a new solution exceeded the original schedule estimates. However, in the long term, the code is far more maintainable, and that must be the primary goal. The lesson is that even careful review is not always sufficient, and engineers must know when to withdraw, regroup and plot a new strategy, incorporating the knowledge gained.

9. Failing Inspection

Just because a building inspector looks at a house and grants a certificate of occupancy does not mean that all of the work was performed perfectly. You may find that although a gas line is installed correctly and up to code, its placement is off when you actually attempt to hook up the clothes dryer. Or that although the walls look straight, they are noticeably crooked when furniture is installed. So it is with software: even with testing, reviews, checks and balances, bugs are found because users use the software in unanticipated ways.

One customer reported a thread starvation problem when several equal priority tasks performed database operations and a lower priority task performed checkpoints. The problem was that when the checkpoint task wrote a dirty buffer, it set a flag in the buffer indicating that I/O was in progress. However, the checkpoint task was pre-empted by a higher priority database task. If the database task needed to access the page in the buffer being written by the checkpoint task, starvation occurred. However, Sleepycat found and fixed exactly this starvation issue five years ago, in 1997. In our fix, the database task, seeing that I/O was in progress, released all of its locks and relinquished the CPU, letting the checkpoint thread continue. So, why was this happening again?

Unfortunately, the original fix only worked on systems where the competing tasks were of equal priority or the system was lightly loaded. This report was on a true real-time system, where all high priority database tasks were given the opportunity to run when the one task yielded the processor, and so the checkpoint task was never able to run. Understanding the fundamental problem was challenging because the fix was already in the code and we needed to expand our thinking to the real-time space to understand why it still failed. The lesson is that software is intimately related to its environment, and reliable software must be both general in nature and flexible, as the user's environment will always differ from the developer's environment.

Another recent challenging problem occurred while running our test suite on an embedded system. A handful of tests were taking an assertion while acquiring a self-blocking mutex lock, because the locking code was unexpectedly returning an `EDEADLK` error. This particular code is one of the few places where we use self-blocking mutexes.

In DB, the code to allocate and initialize a mutex takes an argument for flags. Some of the flags affect the mutex, such as the one indicating that this is a self-blocking mutex. Some affect the allocation code such as one indicating whether the shared memory region (where the mutex is allocated) needs locking or is already locked by the calling function. Therefore, the mutex code looked like this:

```
if (we need a new mutex){
   __db_mutex_setup(..., &m,
      (SELF_BLOCK |
      is_locked ? NO_LOCK : 0));
   MUTEX_LOCK(m);
}
MUTEX_LOCK(m);
```

It was the second call to MUTEX_LOCK that returned the EDEADLK instead of blocking as expected. So, why was this failing, and only on this one system and nowhere else? The possibilities included:

1. This system used `pthread` mutexes. Most systems we have use test-and-set mutexes. Perhaps there was a bug in our Pthread mutex code.

2. Since self-blocking mutexes were not frequently used, perhaps we were hitting a bug in the system's `pthread` implementation.

3. It was something else.

Given that the test suite was only failing on this system and no other in this way, our tendency was to think option #2 was the most likely cause. Option #1 was a possibility but that code is extremely stable in DB and has been virtually unchanged for years.

Fortunately, we have a multi-threaded mutex test application that directly calls the DB mutex code. After easily porting that to the embedded system, and many successful runs, we concluded that the mutex code worked as expected (both DB's and the system's) and the failure must be due to option #3 above and we were almost back where we started. Additional, fairly painful, debugging yielded the true bug, and it is in the code snippet above. The bug was that the SELF_BLOCK flag was never getting passed into the setup function, due to a misplaced parenthesis and different precedence. The correct code must read:

```
if (we need a new mutex){
   __db_mutex_setup(..., &m,
       SELF_BLOCK |
      (is_locked ? NO_LOCK : 0));
   MUTEX_LOCK(m);
}
MUTEX_LOCK(m);
```

Debugging on this particular embedded platform is not very easy. So working through this problem was more difficult than it would normally be. After working through this problem a few questions needed to be answered.

1. **Why did this problem only show itself on this one system and nowhere else?** Almost all other systems use test-and-set mutexes, which don't use the pthread code. The test-and-set code ignores the SELF_BLOCK flag. The other system we have using `pthread` mutexes used a different code path.

2. **What did we learn?** The lessons learned here are that it is important to run the test suite on every system possible and follow up vigorously with all problems. A few times during this debugging, which took a couple of days, we were ready to simply assume it was a system problem and move on. Thankfully we resisted that urge.

10. Preparing to Build the Next House

Just as builders should review their experiences after completing developments, software developers must also review their projects. If a builder built in a town with strict codes and the builder successfully adapted to those codes, s/he might consider retaining those practices even in the face of more lenient codes in another town. Doing so will likely garner the builder a reputation for building a high-quality product, and it's always simpler to have one process in place than many. In software, our lesson for coding standards is that high-quality, clean code is a matter of constant, diligent practice. You cannot start sloppy and end clean. You need to incorporate the cleanliness from the beginning.

A builder may find that of the handful of house designs, several are never chosen by customers and new, expanded designs are required. In our software, when the subsystem no longer meets the needs and requirements of current problems, it is time to consider replacing that subsystem, knowing the experience that went into it in the first place. You can tell it is time to consider such a measure if you are not making reasonable progress on bug fixing or new features. If the subsystem has gotten too complicated, requirements have changed, or the design was wrong, you will usually find that normal maintenance becomes increasingly difficult. That is a good sign that it is time to start over. A common life-cycle for software is the gradual, constant addition of code, until the software is so unmaintainable that it can no longer be supported or developed except through the efforts of armies of low-paid, brute-force testers. Eventually, everyone throws up their hands, retires the product, and starts with a clean sheet of paper. This evolution doesn't have to happen: if the software has been well-maintained and re-architected as needed, with old features being discarded and removed from the code as new ones are added, the code base may never have to be rewritten from scratch [10]. Absolute compatibility will not happen, of course, but necessary compatibility will.

11. How to Become a Builder

Someone who has built a treehouse or a shed might decide they really want to become a builder and build a house. While s/he may know how to make a hammer meet a nail, there are larger issues to deal with in order to be successful in that transition.

We have several suggestions for researchers who want their code widely used. There should be no surprises in this list; it simply summarizes the points that have been made throughout the paper.

1. Write documentation. The source code is not self-documenting and code written by a myriad of independent students or researchers needs to be documented in a traditional sense. Users attempting to use your software need written guidance and specific instructions.

2. Choose and enforce a coding standard. It will teach students an important lesson early on, and it will make life easier for all students and others coming later. Also, people using your code will be able to read it and navigate through it, because the code is consistent and predictable.

3. Invest in release engineering so that a user can easily download your code and run it, anywhere. That can mean building binaries, using Autoconf, or building your own configuration system.

4. Be willing to answer questions and help people get over the initial hurdles of using your software. What is obvious to you because you've thought about it for the last several years may not be obvious to someone else immediately. There is a tendency to become irritated with trivial questions and assume that the "intelligent" users wouldn't ask such things. This is an enormous mistake; while these trivial errors might indicate that the user has not read the manual, more frequently, they indicate that something is not documented or is documented, but confusing. Treat user questions as you would paper reviews and ask yourself, "What information does this person need that s/he was unable to get from our documentation?" Being responsive to user input and questions will require personnel who are responsible for the task.

5. Clean up the code after you've finished writing a paper. The act of publishing tends to leave the code littered with quick hacks that were necessary to run the right tests at a particular time.

The constant tension in all this is that academia places little value on these activities, so it is difficult for academic researchers to devote the time, energy, and finances to this process. The tension might be somewhat less in an industrial setting, but industrial research groups do not always have the personnel or resources to devote to the process either.

12. References

[1]. AT&T, DBM(3X), Unix Programmers Manual, Seventh Edition, Volume 1, January 1979.

[2]. Berkeley Software Distribution, NDBM(3), 4.3BSD Unix Programmers Manual, University of California, Berkeley, 1986.

[3]. Berkeley Software Distribution, DB(3), 4.4BSD Unix Programmers Manual, University of California, Berkeley, 1994.

[4]. RTI Health, Social, and Economics Research, The Economic Impacts of Inadequate Infrastructure for Software Testing, RTI Project Number 7007l.011, NIST Planning Report-02-3, May, 2002, `http://www.nist.gov/director/prog-ofc/report02-3.pdf`.

[5]. Seltzer, M., Yigit, O., A New Hashing Package for UNIX, Proceedings of the 1991 Winter USENIX Technical Conference, Dallas, TX, January 1991, 173–184.

[6]. Seltzer, M., Olson, M., LIBTP: Portable, Modular Transactions for UNIX, Proceedings 1992 Winter USENIX Conference, San Francisco, CA, January 1992, 9–26.

[7]. Software Engineering Institute, "Building High Performance Teams using The Team Software Process and Personal Software Process," Carnegie Mellon Uni-

versity, `http://www.sei.cmu.edu/tsp`, January 20, 2002.

[8]. Software Engineering Institute, “The Team Software Process,” Carnegie Mellon University, January 20, 2002, `http://www.sei.cmu.edu/tsp/tsp.html`.

[9]. Spolsky, J., Things You Should Never Do, Part I, Joel on Software, Apr 6, 2000, `http://www.joelonsoftware.com/articles/fog0000000069.html`.

[10]. Spolsky, J., Good Software Takes 10 Years, Get Used to It, Joel on Software, July 21, 2001, `http://www.joelonsoftware.com/articles/`fog0000000017.html.

[11]. ISO/IEC 9899:1999: Programming languages — C

[12]. ISO/IEC 9945-1:1996: Information technology — Portable Operating System Interface (POSIX) — Part 1: System Application Program Interface (API) [C Language].