Extreme Makeover: Bending the Rules to Reduce Risk Rewriting Complex Systems

Sharon Johnson[1], Jia Mao[2], Eric Nickell[1], and Ian Smith[1]

[1] Palo Alto Research Center, 3333 Coyote Hill Road
Palo Alto, CA 94304
{sjohnson,nickell,iansmith}@parc.com
[2] Computer Science and Engineering, University of California, San Diego
La Jolla, CA 92093-0114
jiamao@cs.ucsd.edu

Abstract. We describe our experience using XP to reimplement sophisticated, high-performance imaging software in a research environment. We focus especially on practices we used to derive value from the existing software, notably *reimplementation by ransacking* and *conversion as learning*. Our experience suggests that some of the classic 12 practices which define XP should be adjusted when there is a existing, well-structured system to serve as a guide.

1 Introduction

Approaching Christmas, 2001, we faced a large and daunting problem. We needed to reimplement a large, complex piece of software that has been in production use for a decade – the Cedar Imager [1, 2]. Reengineering posed new challenges for our understanding of XP. In particular, the roles of customer and programmer needed to change to deal with the fact that the existing system specified a large percentage of the requirements and contained numerous algorithms which could (and in some cases must) be mined by the developers.

The Cedar Imager is a large system which manages the presentation of graphic images from high-level graphic shapes and fonts using abstract color models. It creates rasters of bits or bytes or similar structures for transmission to output devices. Conceived and implemented in the early 1980s at PARC's Computer Science Lab, it was first used as the imaging engine for the Cedar experimental programming environment [3,4,5]. Today, the Cedar Imager is at the core of high-performance printer controllers for monochrome and full color printers.

Developed and supported in a research setting, the Cedar Imager has been of interest to product organizations for more than a decade because of its performance and capability. However, product organizations want the ability to adapt the imaging software themselves and are reluctant to invest in developing and supporting products not written in a commercially-supported language. It was felt that PARC's imaging technology would be more valuable if it were cast in a modern, commercial language, such as Java, but only if the Imager's existing advantages could be retained.

Our project needed to determine whether it is possible to rewrite the Imager into a modern language with widespread industry support and that would retain the strengths of the existing Imager – performance, modularity, and easy-to-read source code. A consistent concern was whether an imager written in Java could provide performance comparable to the Cedar Imager.

Initial work on porting the Imager to Java began in late 2001, at a time when XP was beginning to infect PARC's Computer Science Laboratory [6]. Although unit testing was incorporated from the beginning, no other XP practices were implemented during the project's first seven months. By the end of this period, some of the core functionality was in place: A Java Imager that provided basic imaging facilities that produced gray bytemaps as output, and a decomposer for Adobe's *Portable Document Format* (PDF) [7] which could parse the overall document structure for hand-constructed PDF[1].

2 Our XP Setting

In June, 2002, with the addition of a summer intern, we decided to adopt XP practices wherever possible. Our hope was to gain experience with XP during a 3 month pilot, and to see if pair programming could be used effectively with people who were only going to be involved with the project for a few months. We used one-week iterations, and six hours of pair programming each day.

The XP team consisted of four people: The full-time researcher [Nickell] and one summer intern [Mao] were the development team, while a research manager [Johnson] and an internal XP "coach" [Smith] served as customer surrogates. Our goal for the summer, as well as for the overall project, was to determine the viability of a Java Imager as a future substitute for the Cedar Imager, and to prepare an effective demonstration of this.

3 Our XP Experience

In this section we detail some of our planning game experiences, for the benefit of practitioners who may wish to compare our experience to their own.

During the first four one-week iterations, the tasks picked by our customer surrogates focused on adding basic functionality to the imager and to the PDF decomposer, and on narrowing the focus for a proposed technology demonstration near year end. Because of sharp customer focus, we spent the first few iterations adding the basic functionality needed by that demonstraction, and integrating our software into an existing printing architecture on a color printer. By the end of iteration #4, we could print a PDF document containing colored text on this system.

The planning meetings for iterations 5-7 all involved some conflict between the developers and the customers. By this time, the team had selected some target PDF documents that would be used for a demo. While everyone agreed that risk reduction and containment was a goal, the developers and customers were seeing the primary risk as coming from different directions. Each week, the customers wanted to make progress on the technical area with the largest remaining technical problems, so that the developers were working in very different domains each week, each algorithmi-

[1] Since we had no existing PDF decomposer in Cedar, we used standard XP practices to develop a PDF decomposer to drive our Java version of the Imager.

cally complex. The developers, however, felt that there was risk in having the developers switch between different areas of the code too rapidly. They were in favor of implementing not only basic functionality, but also addressing performance issues in that area before switching to another task. The customers won.

After iteration #7, we were printing sample PDF documents with graphic shapes, strokes, and text, but we had also collected data showing that we were running about 20 times slower than the Cedar Imager.

In the planning meeting for iteration #8, we resurrected the debate of performance vs next-piece of functionality. Keep in mind that "at comparable performance" is a critical part of the project's research question. This time around, the team agreed that low performance was now the greatest threat to the project. As such, the next three iterations, 8-10, were focused on bringing the Java Imager's performance in line with the Cedar Imager's for equivalent functionality. In iteration #8, we explored a variety of optimizations, noting any performance increases and our subjective sense how the optimization affected code smell. During iteration #9, we began selecting which optimizations to fold back into the main branch. In general, we only accepted optimizations which boosted performance by at least 20% and were localized to a couple methods in a single class or which could be isolated in a separate class.

We should note that the full-time developer went on vacation during this period, leaving the summer intern to implement the various optimizations. We still gained much of the advantages of pair programming by pairing and reviewing the selected optimizations as they were folded into the main branch. During iteration #10, we made the difficult decision to replace a compiler-compiler-based parser [8] with a hand-built parser system, to avoid a performance bottleneck inherent in the code generated by the compiler-compiler. By the end of this iteration, Java Imager performance was on par with Cedar Imager performance for our target pages.

During our final iterations, #11-12, with performance concerns addressed for the time being, the team assessed the features that would be needed to print the selected target documents for the year-end demo, and the iterations focused on implementing just those features.

4 Our Re-engineering Practice

Over time, we developed several practices to help us make the most of the existing Cedar code base while developing the Java Imager. Some of these practices were founded on assumptions we made about the Cedar Imager and its developers:

- The developers were world-class computer scientists who usually found the simplest, most elegant partitioning of the problem.
- The existing Cedar Imager core has been in constant use for nearly two decades, and the developers have good mechanisms to track and remove bugs. In our experience, it is remarkably bug-free.
- The Cedar code had been heavily optimized over a long period. Some of the optimizations were largely irrelevant for today's imaging requirements. Other optimizations were superseded at a higher architecture level and were no longer hot spots, but were left in place.
- Some code, particularly some optimizations, assumed bit-level access to the underlying memory, or relied on the ability to do pointer arithmetic. These were infrequent.

- Software that would have been implemented in Java with a small class hierarchy was implemented in Cedar with a variant record structure. This usually meant that pieces of each Cedar procedure in a module were distributed across two or more Java classes.

Practice 1: Code Size Informing Technical Estimates

During the planning game, the developers used relative sizes of Cedar modules to help us estimate time to complete tasks, as well as perceived complexity and familiarity.

Practice 2: Using an Existing System to Create Tests

When reimplementing a complex algorithm, we frequently used the existing system to compute results that we would then use to test the new system. For example, when we integrated code to rasterize embedded fonts on demand for given point sizes, device resolutions, and affine transforms, we wanted to test that the correct bits were being generated. Creating our own easy-to-test embedded font would be painful, as well as using an existing font and hand-computing the correct output pixels. Instead, we wrote code to have the Cedar Imager rasterize a sample font with different parameters, and output the results as text that we cut and pasted into an appropriate data structure in the test code.

Practice 3: Reimplementation by Ransacking

When one of an iteration's tasks was to re-implement functionality already present in the Cedar Imager, we usually neither slavishly translated each line of Cedar into the corresponding Java nor chose to ignore the Cedar code and do a clean sheet implementation. Instead, we chose something in the middle, reimplementation by ransacking.

In this approach, we would use the Cedar code as an example of how some smart people had implemented this same functionality in another setting. We would start by extending our PDF capabilities to the point that it was ready to call the Imager, using standard test-driven development. At this point we would open or create the Java class where we wanted to put this new capability, and paste in the Cedar code as comments. Next, we would identify the Cedar procedure which could provide the needed functionality, and reimplement that procedure as a Java method. At this point, assuming we understood the functionality covered by the Cedar procedure/Java method-to-be, we would translate line-by-line if that would suffice. Frequently, we would encounter a Cedar element, such as an inner procedure, that would require that we pause and consider our reimplementation options. By the time we had finished reimplementing this one procedure, we had calls from our new method to one or more other methods that also would need reimplementation, so we would repeat this process with the other procedures until the class compiled.

A frequent pattern in the Cedar was code that detected special cases which could be optimized, followed by code which implemented the general case. Here, we would only implement the general case. We also discarded other optimizations, such as object pooling.

To any method created in this way, we inserted an initial line, `assert false: “no test written”`. This line would not be removed until we were convinced that the test code had an adequate test for that method’s functionality. Also, though the Cedar language provides access modifiers for its procedures very like those for Java methods, we chose to ignore these, and instead made each reimplemented method private until we were compelled to expose it.

If a section was complex enough that we had to take a significant amount of time understanding it before we could reimplement it, then we encoded that understanding as a comment in the code. We also felt free to refactor and rename if that contributed to better understanding.

At this stage, we now have a system which compiles, but if we attempt to run the entire test suite, it fails, because the test code for the PDF/Imager bridge triggers a call to a newly-implemented method whose first line is an “assert false”. At this point, we would shift from *reimplementation by ransacking* to something much more akin to *test driven development*, writing tests, removing the “assert false” and implementing missing code.

Practice 4: Conversion as Learning

Occasionally, as with iteration #6 when we were adding stroking to the Imager, we had a general understanding of the high-level task. When we examined the Cedar code, we could not immediately gain an understanding of the algorithms it was using. In these cases, we came to use *conversion as learning*. To some extent, *conversion as learning* is the converse of *reimplementation by ransacking*. In *reimplementation by ransacking*, we use our existing understanding of the task to guide our plundering of the Cedar code, working from the top down. In *conversion as learning*, we work our way through the Cedar code to build a bottom-up understanding of an algorithm to implement. Note that both approaches assume that the Cedar developers knew what they were doing.

We would choose some Cedar procedure to start converting, usually starting where we thought we understood it best. We would add assertions and comments as we translated, especially commenting the methods inputs and outputs. Assertions served as comments to us (not to mention future readers) but also helped us quickly revise our understanding if it was violated during testing. Then we would work on translating a related procedure. At some point, we would gain an insight into how the pieces fit together, either for a subsection or as a whole. That new insight would usually drive us to revisit the already-translated code and either recomment or refactor it. We would also have the option, at this point, of continuing with *conversion as learning* or switching back to our normal mode of *reimplementation by ransacking*.

5 Lessons Learned

We chose to adapt some of the XP practices in order to gain XP's over-arching goals. Some in the XP community will consider this a heresy, but if the basic problem of software development is risk [9], then we felt that we reduced the risk of increased cost or development time by modifying the practices to account for the fact that certain types of knowledge were embedded in the existing system. Thus, we needed to find ways to mine that understanding without impairing the integrity of the new system.

Lesson 1: Life without Metaphor

Instead of using a metaphor to guide all development and to guide communication between developers and customers, we used the existing system. In our case, we discussed features in terms of what could be imaged onto a page, using the existing system as a reference. To do this, the developers had to learn to articulate abstract features, such as "fill path", in terms of their visual impact on a page.

Lesson 2: Not Always the Simplest

While the developers did try to remove extra complexity as it was found, there were notable exceptions. This extra complexity arose from straightforward translation of Cedar code which did more than strictly required by the current level of functionality in the Java Imager. This extra complexity could be represented by a more general algorithm than we yet needed, or a more complex data structure. At these points, we were faced with a choice: Should we leave the excess complexity in, or strip it out? Classic XP would suggest that we remove this complexity before moving on. But if we had a task in hand for the current iteration that would require the reintroduction of the complexity, or if we were reasonably sure it would be required in the next, we chose to live with it. More precariously, there were times when our undestanding was not deep enough to know whether the code would be needed in a few weeks or not. Where practical, we would insert an assertion that would prohibit accessing the additional capabilities. This seemed like the simplest compromise between tearing out code we might eventually need, and making sure that there was no functionality without a test.

Lesson 3: Not Always Test First

As mentioned above, we often wrote our tests after writing the code.

1. Ransacking the existing system was much faster than clean-sheet design and allowed us to tackle larger chunks of functionality in a single task. But since some of our understanding was coming from the ransacking process, we were not able to write effective tests until afterward.
2. Test first in classic XP helps a pair of developers design a clean class interface. We were relying in part on the existing system to help us design this. As such, early test construction was not quite as critical.

That said, we still needed a mechanism to give us confidence that our test coverage was adequate for the current system. We could have used a task list. Instead, we chose to hobble methods developed in advance of unit tests by throw an exception as the first statement. We believe there are few, if any, untested features in the Java Imager.

Lesson 4: Using XP with an Intern

Interns usually work on a narrowly circumscribed or separable project or demo, or they work on a forked branch of a main project, or their code is carefully screened by a host before it is checked in. All of these approaches make the cost of integrating the intern's work into the larger project more expensive. One of our motivators for adopting XP was to see if we could use a summer intern effectively as a programming partner, deriving greater value long-term from their work. Our findings:

- Our permanent researcher [Nickell] perceived an immediate jump in his productivity, even during the first 2-3 weeks that our intern [Mao] was navigating and not driving. We suggest some reasons for that below.
- Even from the first day, Mao provided a second pair of eyes for typos, mistaken divergence from the Cedar source, or while debugging. By the third week, she was doing some of the driving.
- Pairing and continuous integration meant that when the intern was driving, she was able to receive prompt feedback when writing less-than-ideal code, rather than having the code buried until an end-of-summer code review.
- Pairing was a natural mechanism to transfer knowledge about the current state of the software with little cost.

We suggest that the following effects may have contributed to the perceived productivity increase even in the first two weeks:

- **Gym Effect:** Like meeting a friend at the gym, agreeing to meet at a certain time, and setting aside 6 hours each day to program with another person increased the actual hours per week writing software.
- **Email Effect:** When a solo programmer hits a difficult problem, or an intransigent bug, there is a tendency to switch tasks. There is email to be checked, or a phone call to be made. While pair programming, this would be socially awkward, so there instead the two begin discussing the problem, which often leads to a solution or an experiment.
- **Hawthorne Effect:** [10]: Workers become more productive simply by being observed. When we are pair programming, we are being observed by a partner. We suspect there is a natural tendency to want to appear astute and knowledgeable, and so we apply ourselves.
- **Meeting Effect:** In a work context where XP is rare, two people working together culturally constitute a meeting. A third party with a question for one of the pair is more likely to tread lightly, either deferring the question until later, or making a brief interruption to negotiate a later time for discussion. Our experience was that development was less interrupted.

In all, our assessment was that Mao contributed significantly to this project, more than would have been possible without using XP.

6 Conclusions

We found XP to be a powerful approach when reimplementing a complex, well-structured software system. To the degree that the expertise embedded in the existing system has value and is exploitable, it can be extracted rather than being recreated in the ongoing interactions between customers and developers. To use XP most effectively in this setting, we found it necessary to modify three core XP practices.

Acknowledgments

We would like to thank our management, Richard Bruce and Richard Burton, for allowing us to experiment with XP in this project and borrow an XP coach. We also wish to express our thanks to the Cedar, Cedar Imager, and Xerox DocuPrint developers for their excellent software and helpful discussions.

References

1. Bhushan, Abhay M. Plass. *The Interpress Page and Document Description Language*. IEEE Computer, 19(6):72-77, 1986.
2. Warnock, John and Wyatt, Douglas K., *A Device Independent Graphics Imaging Model for Use with Raster Devices*, Computer Graphics16, 3.,1982.
3. Deutsch, L. Peter, and Taft, Edward. *Requirements for an Experimental Programming Environment*, Research Report CSL-80-10. Palo Alto, California: Xerox PARC, 1980.
4. Teitelman, W. *The Cedar Programming Environment: A Midterm Report and Examination*, Research Report CSL-83-1. Palo Alto, California: Xerox PARC, 1984.
5. Lampson, B. *A Description of the Cedar Language: A Cedar Language Reference Manual*. PARC Technical Report 83-15, Xerox Corporation, Palo Alto CA., 1983.
6. Bellotti, V., Burton R., Ducheneaut N., Howard M., Neuwirth, C., Smith, I. *XP In A Research Lab: The Hunt For Strategic Value*. In Proc. Of XP 2002 (Alghero, Sardinia, Italy, May 2002), 56-61.
7. Adobe Systems Incorporated. *PDF Reference, 3rd ed., version 1.4.* Addison-Wesley, Boston, MA, 2001.
8. WebGAIN Web Site. On-line at: http://www.webgain.com/products/java_cc/
9. Beck, K. *Extreme Programming Explained*. Reading, Massachusetts, Addison Wesley, 2000.
10. Mayo, E. *The human problems of an industrial civilization* (New York: MacMillan) ch.3, 1933.