

Pair Programming on the C3 Project

Jim Haungs, Oracle Corporation

Chrysler's Comprehensive Compensation (C3) project was one of the first large-scale IT projects on which Extreme Programming precepts were attempted. (For an overview of XP and the C3 project, see Kent Beck's "Embracing Change with Extreme Programming," *Computer*, Oct. 1999, pp. 70-77, and Chet Hendrickson's sidebar, "DaimlerChrysler: The Best Team in the World," p. 75.) As XP was being invented, we didn't really think of it as a theoretical or methodological breakthrough; it was simply an opportunity to get the job done. The "theory" came about later because of the practice's success. In my opinion, XP is not a theory, but a cogent descriptive body of successful praxis.

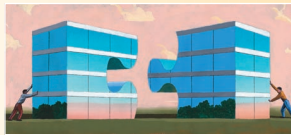
Some people consider pair programming, an XP technique, to be difficult, unworkable, or downright weird. After all, what does the workstation represent, if not finally having your own computer? The idea of sharing your machine with someone seems like a lot of bother without much reward. How can you talk if you're trying to think? And how can you think if you first have to explain what you're thinking about?

Having participated on the C3 project as a performance-tuning consultant beginning in 1996, I can attest to the success of pair programming. At the same time, I can show instances where it did not work or was not emphasized in this early XP project. In a sense, this pragmatism is the strength of XP: Use a technique where it works, ignore it where it doesn't. XP has never been prescribed as a panacea.

COBOL? NO THANKS

The C3 project was innovative on a number of levels, not the least of which

was the language and technology used on the project. The system was implemented in Smalltalk atop a GemStone object database. The choice of these two technologies had a significant impact on the course of the project. While generally good enough, Smalltalk is not the world's fastest language, and this version of GemStone had a particularly slow interpretive implementation of the language.



XP's pair programming helps programmers synthesize their individual expertise into an effective combination.

If the project was to scale up to pay hundreds of thousands of people every week, certain parts of the system had to run a lot faster than they did. In the first versions of the system, it took almost a minute to compute the payroll for a single person. Scaling to the requisite throughput required an order of magnitude performance increase.

Not all of the system was written in Smalltalk—a Cobol program running under Unix did the payroll tax computation. An analysis of the processing time revealed that computing an employee's payroll taxes took the majority of the runtime: about 30 seconds per employee. A compute server running on a different Unix machine performed these calculations. The Smalltalk part of the system wrote a text file containing the payroll numbers for an individual to an NFS-mounted file system. It then sent a message to a process running on the compute

server, which looked for the text file, read it, parsed it into a format suitable for input to the tax library, and called the library subroutine, which computed the tax. The compute server then wrote the output back to another file and informed the Smalltalk program that it was finished, whereupon Smalltalk read the output file, parsed it, and stored the computed tax numbers in the payroll objects.

Fortunately for the tuning effort, the bottleneck stemmed from how the tax computation was invoked, not the computation itself. Writing the input and reading the output files over the network, invoking a new Unix process on the server for every employee, and parsing the input and output files all took too much time. Moreover, we couldn't parallelize the payroll computation because the invocation mechanism effectively serialized the tax portion of the payroll computation.

Upon analysis of the Cobol programs, we ascertained that the tax library took a single parameter—a pointer reference to a large buffer. The buffer contained the input in one area and space for the output in another. A wrapper program read the input file, converted it to the buffer format expected by the tax library, and invoked the tax subroutine, which wrote the results back into the same buffer. Then the wrapper program wrote the output to a text file.

Because the actual computation was fairly fast, though hobbled by the invocation mechanism, this architecture was an ideal candidate for conversion from a compute server architecture to a simple subroutine call from Smalltalk. To match the Smalltalk calling conventions to those of the Cobol program, we wrote a C wrapper (technically a thunk) to convert the Smalltalk object pointers to Cobol parameters and then pass the parameters to the Cobol program.

The analysis task centered around how Cobol accessed “by reference” parameters and how the Cobol compiler could be coerced into generating position-independent machine code so the tax routines could be linked dynamically from a shared library instead of statically into the Smalltalk runtime. We also looked at how and when the Cobol runtime library was initialized and how the buffer could be allocated, and, if possible, reused between calls.

After several weeks of detail work, we successfully called the Cobol library from Smalltalk, reducing the tax computation time from 30 seconds to 14 milliseconds. The buffer was allocated once upon startup and reused on subsequent calls. We never got the tax routines to load dynamically, although later versions of the Cobol compiler supported this feature. As a result, we had to relink the Smalltalk runtime whenever the tax library changed, which was usually once a year. Although inelegant, the dramatic speedup compensated for this unfortunate limitation.

Did this project afford opportunities for pair programming? Not really. Most of our time was spent in activities such as support calls discussing arcana like register calling conventions, writing the specialized C code to call the library, or stepping through assembly code in the debugger. The XP practice that stood out in this effort was its emphasis on testing. Having dozens of unit tests to check the tax computation allowed us to set up parallel systems, one using the old way and one the new. The results could be trivially compared—the unit tests either worked or they didn’t. When they did, they provided exact answers calculated more than a thousand times faster.

THE PROFILING TOOL

After solving the worst performance problem, we began analyzing the Smalltalk parts of the system. This proved more complicated than expected. At the time, VisualWorks Smalltalk and GemStone Smalltalk were subtly different languages. VisualWorks dealt strictly with ephemeral objects, whereas GemStone concerned itself with persistent ones. The tuning requirements of the two are quite different. Both store objects in Collections, such as Sets or Dictionaries, but GemStone col-

lections can optionally have indexes like a relational database. Moreover, GemStone must deal with locks and concurrency, while VisualWorks does not.

The C3 team chose to develop the program entirely in VisualWorks and then load the Smalltalk code into the GemStone database. Places where the languages differed slightly were hidden behind a layer that abstracted the differ-

We could profile part of the system, change it, version the changes, and profile it again.

ences. Usually, this meant adding missing methods to one or the other environment. The team avoided using features that existed in one environment but could not be mirrored in the other.

Both VisualWorks and GemStone contain profiling tools for tracking both CPU time and allocated space. However, the tool implementations are very different. Rich Garzaniti and I built a profiling tool that disguised the complexities of performance measurement behind a very simple user interface, built around a 2×2 matrix: speed versus space on one axis and VisualWorks versus GemStone on the other. We embedded all the necessary profiling hooks in either the runtime systems or the application. With the press of a button, we could profile on any axis of the matrix. The tool used the low-level profiling primitives of the respective systems to collect the profiling statistics, then presented them in a standard form regardless of where and how they were generated.

The results of each profiling run were saved persistently in the GemStone database for later analysis and comparison. Thus, we could profile part of the system, change it, version the changes, and profile it again. If the results were not better, we could instantly back out the changes. We could keep the persistent statistics for weeks and chart how the system performance improved over time.

Pair programming on this project was very effective. Although his background was in accounting—and hacking around in the low-level profiling primitives was

complicated—Rich was very interested in how the profiling tools worked. He was instrumental in producing a simple and elegant user interface, which belied the underlying complexity.

Rich’s knowledge and mine were both deep, but in different areas: the application domain versus the language implementations. Rich knew what data the team needed and how to present it, and I knew how to obtain it. For example, Rich had the idea to save the profiling results in the database. The implementation was difficult, but once the idea was put forth, users found it quite elegant and the productivity improvement was well worth the implementation cost. I had the idea of the 2×2 matrix because I knew that the presentation of the data did not depend on how it was generated.

Pair programming forces you to make explicit your implicit assumptions. Our combined efforts produced a tool that was much greater than the sum of its parts. We are both particularly proud of the user interface’s elegance and simplicity. Its daily use enabled the C3 system to scale from monthly to weekly and finally to hourly payroll computations for hundreds of thousands of employees.

Many people throughout the C3 project used the profiling tool to speed up parts of the system, and it was modified many times to compensate for changes as the underlying Smalltalk implementations were upgraded. Having a consultant build such a tool and then leave might be good for the consultant’s job security, but it’s bad for the project. Sharing the implementation knowledge and passing it on to other project members contributed tangibly to the long-term health of the project, and to the excellent performance of the software. ✨

Jim Haungs is a principal member of the technical staff at Oracle Corporation. Contact him at Jim.Haungs@oracle.com.

Editor: Michael J. Lutz, Rochester Institute of Technology, Department of Computer Science, 102 Lomb Memorial Drive, Rochester NY 14623; (phone) +1 716 465-2909; (fax) +1 716 475 7100; mjl@cs.rit.edu