OBSERVATIONS ON THE EVOLUTION OF A SOFTWARE SYSTEM

by

Eric Allman and Michael Stonebraker

Memorandum No. UCB/ERL M82/59

2 June 1982

ELECTRONICS RESEARCH LABORATORY

This work was supported by the Air Force Office of Scientific Research under grant 78-3596, and Naval Electronics Systems Command Contract N00039-76-C-0022.

Observations on the Evolution of a Software System

Eric Allman and Michael Stonebraker, University of California, Berkeley

The Ingres data base system[1,2] encompasses about 75,000 lines of code in the programming language "C"[3] and runs on top of the Unix operating system. Over the past six years, Ingres has evolved into a functionally complete and usable prototype. Development required 25 to 30 programmer-years by a total of 19 people, and the system is now in use at over 125 sites around the world.

In this article we will attempt to answer a question that we are often asked: "How did you manage to get a large software system to work in a university environment?" A chronology of the project and the major technical mistakes have been reported elsewhere,[2] so we will concentrate on the software engineering process and what we have learned about it. Although our experience is in a research environment, we believe that many of these lessons can be applied to most software development efforts.

Chronology

The Ingres experience can be divided into three periods that can be examined individually.

Initial design and implementation (1974-1976). The initial goal was to build a functional relational data base system. During this period we wrote about 60,000 lines of code. The attitude of the design team was to "make the system work" regardless of the methods used.

The project developed out of a graduate seminar; hence, the initial programmers were all graduate students. They were organized as a chief programmer team of four programmers plus a lead programmer. The project directors, Michael Stonebraker and Eugene Wong, acted largely as "creative consultants." Their role was to define the major structure of the system and determine general strategy. Although they were the final arbitrators on technical matters, most problems were handled by the chief programmer.

The role of the chief programmer was first to write code and second to arbitrate conflicts and supervise global design. Since the PDP-11 computer on which implementation began had a 64K-byte address space limitation, Ingres clearly had to run as several Unix processes. The division of the implementation effort among the programmers followed process boundaries. Hence, for the most part, each programmer could code in his own Unix process and have an address space to himself. Minimal cooperation was necessary once the function of each process was finalized and its interface defined. Each programmer could implement the functions in his process any way he wished. Initially, there was no attempt to share subroutines, and even the routines for interprocess communication were customized for each process. Moreover, there was no utilization of software engineering practices such as structured walk-throughs or reading the code produced by others.

The absence of organizational structure directly reflected the multiprocess environment in which no shared code for Ingres existed. This absence of structure below the process level contributed to many later problems.

Making it work (1976-1978). Once an initial prototype was working, we were eager to have people outside the university use it. By mid-1977 there were about ten users, and the sites were all bold, innovative, and sophisticated. The feedback from these initial users was extremely helpful. In addition to receiving useful information about what "real" people wanted to do, we also gained considerable exposure. Many of the comments concerned the awkwardness of the user interface and the absence of helpful error messages.

Much effort went into making improvements to the system during this time. For example, error returns from system calls had not been consistently checked in the initial version. When the disk became full and no new blocks were available for expansion, the data base would be irrevocably corrupted. We spent considerable time fortifying the system against these kinds of events. We also added crash recovery, concurrency control, additional access methods and did substantial algorithm modification to improve performance.

The project continued to be organized as a chief programmer team of three to five people. As the initial collection of programmers left, we replaced them with undergraduates exclusively. The idea was to obtain two to three years of continuous employment in order to justify the long training period (typically one academic quarter). Also, while graduate students were intrigued with the problem of building a data base system from scratch, they found the extension of an existing system less appealing.

During this period the number of installations climbed steadily, and the chief programmer spent more and more time providing user support, usually by telephone. While early feedback was very useful in isolating poorly designed features and uncovering bugs, later interactions focused on misinterpretations concerning the setup instructions, the reference manual, or Unix. The feedback from these later users was of less value. As we gained a reputation, the new users we attracted were frequently less sophisticated and were starting to demand a turnkey system. We began to act less like a research group and more like a software house.

Back to research (1978-). In 1978 it became clear that the Ingres project was soon to accomplish the goals set in 1974. To maintain a climate of intellectual inquiry, the project expanded its goals to include the following:

(1) Build a distributed data base management system. This necessitated extensive data base code to provide consistency control in a distributed environment as well as distributed query processing.

(2) Integrate Arpanet services. These are required to achieve the goal listed above.

(3) Develop operating system extensions to support local networking.[4] These are also required for goal number 1.

(4) Build a data base machine. Given the design philosophy of Muffin,[5] this was a natural outgrowth of distributed data bases. However, it required a local network and a small real-time operating system. No candidate for either was available at the time we began.

(5) Build a new data base programming language, Rigel.[6] This project was undertaken by Lawrence Rowe.

As a result of these decisions, we found ourselves doing development in the areas of language processing, networking, and operating systems. This has created a large increase in the complexity of the Ingres project.

As before, we continue to have a chief programmer and a staff of three to five programmers, augmented by other individuals or groups. The chief programmer team continues to maintain and extend the core of the Ingres system, while networking and language work are carried out by other autonomous groups. This allows the necessary specialization, although it has increased the number of conflicting factions. For example, networking code requires operating system modifications that often interfere with the data base effort. Moreover, Rowe and Stonebraker, who now share the task of general technical direction, occasionally operate at cross purposes. Frequently, there is no clear resolution of the situation; hence, setting priorities and goals has become exceedingly difficult.

Lessons and observations

Setting goals. Our policy has always been to set long-term goals that are nearly unattainable. This policy has led to great intellectual expansion for the participants and appears to help in the recruitment of talented people, which the success of the project ultimately depends on. Nevertheless, it seems crucial to choose achievable short-term targets. This avoids the morale problems related to tasks that appear to go on forever. The decomposition of long-term goals into manageable short-term tasks continues to be the main job of the project directors.

Short-term goals were often set with the full knowledge that the longer-term problem was not fully understood or that a crucial variable (such as crash recovery) was deliberately being ignored. Consequently, many steps were taken that were ultimately wrong, and they were retraced later when the issues were better understood. The alternative is to refrain from development until the problem is well understood. We found that taking any step often helped us find the correct course of action. Also, moving in *some* direction usually resulted in higher project morale than a period of inactivity. In short, it appears more useful to "do something now even if it is ultimately incorrect" than to only attempt things when success is assured.

As a consequence of this philosophy, we take a relaxed view toward discarding code. Throughout the Ingres project, we have repeatedly done complete rewrites of large portions of the system. Whenever the code became top-heavy with patches, or when we learned that it should have been structured differently, we simply "bit the bullet" and rewrote it. Our philosophy has always been that "it is never too late to throw everything away." Although this has proved expensive at times, it usually served us well by eliminating unwieldy pieces of code. However, as the system grew, larger and larger pieces of code came under the scalpel. There came a point at which rewriting bulky and uninteresting pieces of code became impossible because the rewrite was so tedious that it would incur an intolerable morale cost.

When implementing a major system, it would seem wise to plan to build a prototype that will be tested and thrown away. This was the strategy followed by the

System R design team.[7] Only well-understood problems can be properly implemented the first time.

System decomposition. Advantageous system decompositions and well-defined interfaces enhance understandability of the entire system and simplify system construction. Moreover, we found it desirable for each programmer to have a major module to himself so that he can feel like a substantial contributor. Only good system decompositions allow this without having people interfering with each other.

Top-down design is usually suggested as the correct mechanism to achieve this goal. However, we have had frequent difficulty following this seemingly sound advice. Several examples have been presented elsewhere.[2]

Top-down design assumes that the problem is completely understood and that there are no external constraints to contend with. Since we were bound by the maximum size of a PDP-11 address space, we frequently found that a process was not large enough to contain the code for planned functions. When this happened, we were forced to restructure the code. Also, we found that in several areas of the system a clean top-down design incurred an intolerable performance penalty.[2]

Although we had the freedom to begin with a top-down design, thereafter we were restricted to making feasible changes to a running system. In a sense, there is a collection of "next states" to which the software can evolve in the next iteration. Such states are highly constrained by previous (perhaps incorrect) decisions.

For example, in 1977 we added crash recovery to a working system. This entailed identifying all failure patterns and leaving enough "footprints" in the data base so that a recovery utility could correctly clean up after the failure was repaired. This code involves myriad low-level changes to dozens of routines. Restructuring all the low-level routines in order to cleanly add recovery was not even considered because it would have necessitated a large rewrite that was not considered feasible. Rather, crash recovery was inserted incrementally in an ad hoc way.

As the above example illustrates, we were constantly pulled between two points of view: restructuring and rewriting to achieve a clean design in the next iteration, or seeking an ad hoc solution because the cost of the first alternative might be too high.

In summary, top-down design seems to have been effective when requirements were well understood. However, much of the Ingres code evolved as requirements were identified during or following initial implementation. In retrospect, the intuition of the system designers seems to have been the most reliable design technique.

When the Ingres system was initially designed, we expected a multiprocess organization to be highly advantageous. It offered the possibility of parallelism, forced a clean decomposition of function at the top level, and gave each programmer an isolated environment. Unfortunately, we found that multiple processes were fundamentally undesirable. Not only are they hard to reset when errors occur, but they are also inflexible when the top-level control structure inevitably changes. Lastly, repetition of functions is often required. For example, all processes must individually open the files in which system catalog information is kept.

Although Ingres is forced to use multiple processes because of the address space limitations on a PDP-11, these processes are perfectly synchronized; each waits for a successful return from its neighbor before it accepts new work. The 32-bit address space available on VAX computers allowed us to collapse all processes together.

Clean code. By and large, we made continuous and demonstrable progress during the initial stages of the project. We chose to ignore hard problems and write "dirty code." But as the Ingres system grew larger, it became impractical to write badly structured code, since this had a tendency to complicate future debugging and maintenance.

Many programmers lean toward twisted, tricky solutions to problems. Although this may improve efficiency in the short run, we found complex code undesirable. Inevitably, it was hard to debug and maintain. When the original designer of a module departed, leaving it in the hands of a new person, maintenance costs escalated. If "tricks" are truly necessary to meet performance requirements, they should be elevated to the status of carefully documented techniques.

The transition from dirty code to clean code was a painful one. Perhaps we should have started writing clean code from the beginning. However, this would have increased the time required to produce an initial protoype, and a working prototype was essential to establish a reputation. Moreover, at the time, we were unaware of the ultimate pitfalls of dirty code. As a result, we believe that a phase of dirty code was a necessary stage in our evolution, as were the growth pains of cleaning it up.

Coding standards. When a module changes hands, the recipient frequently alters the program to suit his particular style. A certain amount of this is desirable because the new programmer may notice an easier way to do something and make appropriate changes. However, we found that an inordinate amount of time was spent adjusting Ingres code to the personal style of each programmer (e.g., changing the way programs were laid out on line-printer output). In an attempt to restrict this extra editing, we instituted a set of coding standards.

The initial reaction was exceedingly negative. Programmers used to having an address space of their own felt an encroachment on their personal freedom. In spite

of this reaction, we enforced standards that in the end became surprisingly popular. Basically, our programmers had to recognize the importance of making code easier to transfer to new people, and that coding standards were a low price to pay for this advantage. The results met our goal; random changing of program style has all but disappeared, and readability of the system has increased.

When the Ingres project began building a distributed data base system, we left a period of "bulletproofing" the system and entered one of less structured experimentation. At this time, there was a proposal to drop the coding standards. Popular opinion, however, called for them to remain, although ironclad enforcement has disappeared.

Coding standards should be drawn up by a single person to ensure unity of design; however, input should be solicited from all programmers. Once legislated, the standards should be rigidly adhered to.

Documentation. The Ingres source code is well documented and a creditable reference manual exists. However, no documentation for major internal interfaces has ever been written, nor has a guide for new people ever been devised. As a result, there is a long learning curve for new Ingres programmers. In addition, considerable interaction with other project members is required during the training period, and it is almost impossible to make use of transient help. Currently, this is considered a major weakness.

A university is not a software house, and there is little incentive to produce documentation. We probably could have benefited from imposing documentation requirements on the entire staff from the beginning, but the cost of producing substantial internal documentation today is very high.

Proposals have been made for internal documentation in the form of comment blocks preceding programs and procedures. These would include fields such as *name, function, algorithm, parameters, returns, globals, calls, called by,* and *history*. We attempted to use this style of comment block in new Ingres code for two years. The results were mixed. Fields such as *parameters, returns,* and *side effects* seemed useful because they warned the programmer when the semantics of the routine were being changed. However, certain other fields, such as *history,* were generally not kept up to date. Since the usual cycle is edit, compile, test, it is unnatural for the programmer to go back and add a line in the history field, particularly if the change seems small. The final conclusion was that an out-of-date history is worse than none at all.

Fortunately, many fields, e.g., *globals, calls, called by,* and *history,* can be maintained automatically by language processors or source-code control programs. A sound rule is that any information that can be maintained automatically should be; manual maintenance is never as accurate and seldom as convenient. Fields such as *algorithm* are usually just a restatement of the code and should be eliminated in favor of comments interspersed in the code to explain what is happening. Such comments are easier to maintain and are more readily associated with the code they describe.

Tools. In order to maintain some internal documentation automatically, we have recently started using the Source Code Control System.[8] This package automatically keeps track of who makes changes and prompts for descriptive comments about each change. Furthermore, the code for both the old and new versions is available, so backout is possible in case of disaster. The system has proved both effective and popular. It is far superior to comment blocks maintained by hand.

In general, we believe it is almost never a mistake to spend extra time investing in tools. We have made good use of the tools provided in the standard Unix environment, for example the parser generator YACC.[9] In addition, we spent considerable time importing tools such as the Vi text editor,[10] the "C-shell,"[11] and the Source Code Control System. We found these investments well worth the cost.

Support. There is a big difference between making Ingres work ourselves and enabling other people to make it work. The latter requires an enormous effort to eliminate bugs and provide user-level documentation, along with easy-to-follow installation procedures. Perhaps only a third of this total effort is required to get a large system to the stage where *we* can make it work.

Whether we like it or not, the Ingres project is in the support business. With over 125 installations, we get many phone calls that consume the time of key people. Originally, the calls gave valuable feedback about the system; today they tend to be less technical and more administrative. The fear of even more phone calls tended to inhibit changes to later versions of the code because changes inevitably introduce bugs or confuse users. Such a fear is counterproductive in our research environment.

With a system at this level of maturity, it appears necessary to engage a separate support organization if research and development are to continue. Realistically, support should be considered as the system is developed; reliability and maintainability are built in, not added on.

Hardware environment. Morale problems associated with unworkable hardware have been a serious project issue. They not only frustrate the programming team and slow progress, they also consume the time of key people who must cope with or remedy the situation. In retrospect, it is clear that if we had had twice our hardware budget at the right time, it would have been advantageous

to buy all our hardware from one vendor. The Feldman report[12] may help in this regard.

Complexity. By setting high goals and embarking on distributed data base/operating system/network projects, we have created a software environment so complex that the project directors no longer have a good grasp of operational trade-offs. As a result, priorities are extremely difficult to set and often appear inconsistent from week to week. Moreover, the chief programmer and project directors spend a lot of time battling brush fires. This usually means that rational long-range planning is neglected. Consequently, maintenance and conversion efforts have been poorly planned and seem to go on forever. This creates a morale problem for the implementation team. Lastly, the presence of multiple networks and home-brew network hardware has created an unstable machine/operating system environment that amplifies any morale problems. The structure seems in danger of collapse at all levels.

The project directors and chief programmer are overloaded with work and beset with nontechnical details that interfere with useful research. The difficulties sometimes seem insurmountable, and the cost of making progress has become enormous. Other personnel are so bogged down with maintenance/conversion efforts (a new 32-bit machine, three new versions of Unix, and a new version of Ingres) that they see no progress at all.

Earlier we used a throw-away-and-rewrite mechanism to deal with complexity. Recently, we have begun to act more like a software house, although we find the reward structure and support organization impossible to build within a university. We may well have reached a complexity barrier that we will not be able to penetrate with the limitations of the current environment. Certainly, the project appears technically out of control. Even more distressing is the question "Where will the next generation of implementors come from?" The complexity of our environment is such that it taxes undergraduates to the limit. Only juniors and seniors appear to have the background to successfully cope; hence, our previous strategy (hiring freshmen) appears unworkable. On the other hand, masters-level students are available for only one year, and it is widely recognized that implementations are not in the best interests of PhD students.

We are constrained by the limitations of our environment. Since the university supports neither a two-year master of software engineering program nor a significant implementation for a PhD dissertation, we cannot attract the necessary people. Also, without hiring professional managers, we are restricted to a one-level organizational hierarchy, sharply limiting the total size of the implementation.

Perhaps there is a meta-theorem here: "Within our environment, one 75,000-line program can be written, but expanding it to a more complex 150,000-line program is impossible." No doubt all organizations have some fundamental limit; ours may simply be lower than others. To contradict the meta-theorem we would have to find a new way to deal with the complexity.

Conclusions

The Ingres system has gone through several stages of development. The initial stage produced poorly designed code, and little thought was given to maintenance or code sharing. This allowed individuals coding in their own process to exercise creativity to the utmost with a minimum of conflict. Later, to support users and make the system reliable, we were forced to write cleaner code. Recently, we have done less to support users, but the complexity of the system demands that we continue to write clean code.

Our largest mistake was probably in failing to clearly pinpoint the change from prototype to production system. At this point several procedural changes should have been implemented immediately; instead, they were slow to appear and frequently incomplete. We feel that system implementors should clearly identify this point in their development cycle.

In a research environment, any development effort involves problems whose solution is unknown. Under these circumstances, standard software development approaches, e.g. strict top-down design, do not always work, and rewriting code becomes a powerful development tactic. It is a serious mistake to believe that a piece of code is sacred because of the time taken to write it.

Throughout the development of Ingres, we made considerable use of the tools available in Unix. The cost and a few weeks spent setting up and learning to use a major tool are well worthwhile. It is nearly always a mistake to do a task by hand that can probably be performed better by a tool.

References

1. M. Stonebraker et al., "The Design and Implementation of INGRES," *Trans. Database Systems*, Vol. 2, No. 3, Sept. 1976.
2. M. Stonebraker, "Retrospection on a Data Base System," *Trans. Database Systems*, Vol. 5, No. 3, Sept. 1980.
3. D. Ritchie and K. Thompson, "The UNIX Time-Sharing System," *Comm. ACM*, July 1974.
4. L. Rowe and K. Birman, "A Local Network Based on the UNIX Operating System," (submitted for publication), Sept. 1980.

5. M. Stonebraker, "MUFFIN: A Distributed Data Base Machine," *Proc. 1st Int'l Conf. Distributed Computing*, Huntsville, Ala., Oct. 1979.

6. L. Rowe and K. Schoens, "Data Abstraction, Views, and Updates in RIGEL," *Proc. 1979 ACM-SIGMOD Conf. Management of Data*, Boston, Mass., May 1979.

7. M. Astrahan et al., "System R: A Relational Approach to Database Management," *TODS*, Vol. 1, No. 2, pp. 97-137.

8. L. E. Bonanni and A. L. Glasser, "SCCS/PWB User's Manual," Bell Laboratories, Nov. 1977.

9. S. Johnson, "YACC—Yet Another Compiler-Compiler," Computer Science Tech. Report No. 32, Bell Laboratories, Murray Hill, N.J., July 1975.

10. William Joy, "An Introduction to Display Editing with Vi," University of California, Berkeley, Dec. 1979.

11. William Joy, "An Introduction to the C-shell," University of California, Berkeley, Dec. 1979.

12. Jerome A. Feldman and William R. Sutherland, "Rejuvenating Experimental Computer Science—A Report to the National Science Foundation and Others," *Comm. ACM*, Vol. 22, No. 9, Sept. 1979, pp. 497-502.