DOI:10.1145/2461256.2461270

Nancy G. Leveson

Inside Risks

Learning from the Past to Face the Risks of Today

Achieving high-quality safety-critical software requires much more than just rigorous development processes.

As software takes over more and more functions in our increasingly complex and potentially dangerous systems, our software engineering problems are going to increase. The number of failures of large system projects we have been experiencing, particularly government systems, for example,[1–4] is not going to be acceptable. We need to learn from the failures and—even more important—from the successes of the past.

I recently contributed a chapter on software for a forthcoming book on the legacy of the Space Shuttle. A mythology has arisen about the Shuttle software with claims being made about it being "perfect software" and "bug free" or having "zero defects," all of which are untrue. But the overblown claims should not take away from the remarkable achievement by those at NASA and its major contractors (Rockwell, IBM, Rocketdyne, Lockheed Martin, and Draper Labs) and smaller companies such as Intermetrics (later Ares), who put their hearts into a feat that required overcoming what were tremendous technical challenges at that time. They did it using discipline, professionalism, and top-flight management and engineering skills and practices too often missing from today's safety-critical software projects. Much can be learned from this effort that is still applicable to the task of engineering complex software today. This column summarizes some of these lessons.

There can always be differing explanations for success (or failure) and varying emphasis can be placed on the relative importance of the factors involved. Personal biases and experiences are difficult to remove from such an evaluation. But most observers agree that the process and the culture were important factors in the success of the Space Shuttle software as well as the strong oversight, involvement, and control by NASA.

Strict Government Oversight and Learning from the Past

The Shuttle software project benefited from what NASA had learned from earlier spacecraft. Gemini (1965–1966) was the first U.S. manned spacecraft to have a computer onboard. At the time, computer programming was considered an almost incidental activity. Experts wrote the software in low-level, machine-specific assembly languages. Fortran was considered too inefficient

CREDIT TK

for use on real-time systems: The Gemini software development was largely haphazard, undocumented, and highly idiosyncratic.[7]

Computers had little memory at the time and squeezing the desired functions into the available memory became a difficult exercise and placed limits on what could be accomplished. The programmers also discovered that parts of the software were unchanged from mission to mission. To deal with these challenges, the designers introduced modularization of the code by loading only the functions required at that point in time. Another lesson learned was the need for software specifications and simulation programs to validate the guidance equations.

Despite the challenges and the low level of software technology at the time, the Gemini software proved to be highly reliable and useful. NASA realized, however, that the handcrafted, low-level machine code of Gemini would not scale to the complexity of later spacecraft. The problem of how to generate reliable and safe software had to be solved.

NASA used the lessons learned from the Gemini project about modularity, specification, verification, and simulation in producing the more complex Apollo software. In turn, many of the lessons learned from Apollo were the basis for the successful procedures used on the Shuttle.

Computers had little memory at the time and fitting necessary functions into the Apollo computer memory resulted in the abandonment of some features and functions and resulted in the use of tricky programming techniques to save others. The complexity of the resulting code led to difficulty in debugging and verification and therefore to delays. When it appeared that the software would be late, more people were added to the software development process, which simply slowed down the project even more. This basic principle that adding more people to a late project makes it even later is well known now, but it was part of the learning process at that time. Configuration control software was also late, leading to delays in supporting discrepancy reporting.

Another critical mistake, still made too often today, was to take shortcuts in testing when the project started to fall behind schedule. The 1967 Apollo launchpad fire gave everyone time to catch up and fix the software, as later the Challenger and Columbia accidents would for the Shuttle software. The time delay allowed for significant improvements in the software and in the process. Without it, the results may not have been as good.

One of the most important lessons was that software is more difficult to develop than hardware.

To reduce communication problems and control the development process, NASA created a set of control boards that managed all onboard software changes for Apollo. NASA also created a set of reviews for specific points in the development process, now familiar for many government or large company projects today. The review and acceptance process provided for consistent evaluation of the software and controlled changes, which helped to ensure high reliability and inserted much-needed discipline into the software development process. This control board and review structure became much more extensive for the Shuttle.

In the process of constructing and delivering the Apollo software, both NASA and the MIT Instrumentation Lab (which created the software) learned a lot about the principles of software engineering for real-time systems and gained important experience in managing a large real-time software project. These lessons were applied to the Shuttle. One of the most important lessons was that software is more difficult to develop than hardware. As a result:

- Software documentation is crucial.
- Verification must be thorough and proceed through a sequence of steps without skipping any or being rushed to try to save time.
- Requirements must be clearly defined and carefully managed before coding begins and as changes are needed. The dynamic nature of requirements for spacecraft should not be used as an excuse for poor quality.
- Good development plans should be created and followed.
- Experienced personnel should be assigned to a project early, rather than using the start of a project for training inexperienced personnel.
- Software should not be declared complete in order to meet schedules, requiring users to work around errors. Instead, quality should be the primary consideration.

NASA also learned three general and critical lessons: that increased attention to software would be necessary in future manned space programs; software needs the same type of disciplined and rigorous processes used in other engineering disciplines; and quality must be built in from the beginning—quality cannot be added after the software is written.

Using these lessons learned, the software development for Skylab followed strict engineering principles, which were starting to be created at that time in order to change software development from a craft to an engineering discipline. The Skylab program demonstrated that careful management of software development, including strict control of changes, extensive and preplanned verification activities, and the use of adequate development tools, results in high-quality software with high reliability.

Slowly and carefully NASA learned how to develop more complex software for spacecraft. The increasing success was not due simply to individuals learning from their mistakes, but the organization itself identifying the problems and ensuring the successful solutions derived from them were implemented and improved in later projects. Basically, NASA engaged in successful organizational learning.

To ensure these lessons would be applied in the Shuttle software and they would not have to relearn the same lessons from scratch for each spacecraft project, NASA maintained direct control of the Shuttle software rather than ceding control to the Shuttle hardware contractor. The hardware and software

contracts were separated, with the software contractors directly accountable to NASA management. NASA had learned how important software was to the success of the entire program and closely managed the contractors and their development methods.

NASA worked very closely with the contractors and even constructed their own software development "factory" (the Software Production Facility) and test facility (SAIL) at NASA Houston, thus ensuring the highest standards and processes available at the time were used and that every change to human-rated flight software during the long life of the Shuttle was implemented with the same professional attention to detail.

The level of participation and control exercised by NASA is unusual for most government projects today, including many current NASA projects, where privatizing is common. Commercial projects often use outsourcing and subcontracting without careful and detailed oversight of the process.

A Software Development Process that Promoted High Quality

Based on their experiences and learning from past projects, a sophisticated software development process was created for the Shuttle. This development process was a major factor in the software success. Especially important was careful planning before any code was written, including detailed requirements specification; continuous learning and process improvement; a disciplined top-down structured development approach; extensive record keeping and documentation; extensive and realistic testing and code reviews; and detailed standards.

Extensive Planning and Specification. The Shuttle was one of the first spacecraft (and vehicles in general) to use a fly-by-wire flight control system, which created quality and reliability challenges. In response, NASA and its contractors developed a disciplined and structured development process. Increased emphasis was placed on the front end of development, including requirements definition, system design, standards definition, and top-down development.

An important feature of this process was extensive planning before starting to code: NASA controlled the requirements, and NASA and its contractors agreed in great detail on exactly what the code must do, how it should do it, and under what conditions before any code was produced. That commitment was recorded. Using those requirements documents, extremely detailed design documents were produced before a line of code was produced. Nothing was changed in the specifications (requirements or design) without agreement and understanding by everyone involved.

A common excuse used today for not writing requirements first is that the requirements are "unknown" or may change. In fact, in these cases, it is even *more* important to put major effort into upfront requirements analysis and specification. The software requirements for the Shuttle were continually evolving and changing, even after the system became operational and throughout its 30-year lifetime. NASA and its contractors made over 2,000 requirements changes between 1975 and the first flight in 1981. After the first flight, requirements changes continued. The number of changes proposed and implemented required a strict process to be used or chaos would have resulted.

The strategy used to meet the challenge of changing requirements had several components: rigorously maintained requirements documents, using a small group to create the software architecture and interfaces and then ensuring their ideas and theirs alone were implemented (called "maintaining conceptual integrity"), and establishing a requirements analysis group to provide a systems engineering interface between the requirements definition and software implementation worlds. The latter identified requirements and design trade-offs and communicated the implications of the trade-offs to both worlds. This strategy was effective in accommodating changing requirements without significant cost or schedule impacts.

Basically, NASA engaged in successful organizational learning.

All requested changes were submitted to the NASA Shuttle Avionics Software Control Board (SASCB). The SASCB ranked the changes based on program benefits including safety upgrades, performance enhancements, and cost savings. A subset of potential changes were approved for requirements development and placed on the candidate change list. Candidates on the list were evaluated to identify any major issues, risks, and impacts and then detailed size and effort estimates were created.

Once the change was approved and baselined, implementation was controlled through the configuration management system, which identified: the approval status of the change; the affected requirements functions; the code modules to be changed; and the builds (for example, operational increment and flight) for which the changed code was scheduled. Changes were made to the design documentation and the code as well as to other maintenance documentation used to aid traceability.

Detailed design specifications were developed only *after* the requirements specifications. Today in software development (and even touted as desirable by software researchers), design specifications are too often substituted for true requirements specifications or the two are mixed, making requirements analysis and management during development and over the life of the system extremely difficult.

When coding finally did begin, top-down development was the norm, using stubs and frequent builds to ensure interfaces were correctly defined and implemented first, rather than finding interface problems late in development during system testing. No programmer changed the code without changing the specification so the specifications and code always matched.

Due to the size, the complexity, the still-evolving nature of the requirements, and the need for software to help develop and test the Shuttle hardware, NASA and IBM created the software using incremental releases. Each release contained a basic set of capabilities and provided the structure for adding additional functions in later releases. These incremental releases were carefully planned to ensure later additions could be successfully integrated without requiring extensive changes. These planning and specification practices made maintaining software for over 30 years possible without introducing errors when changes were necessary.

Continuous Improvement. Continuous improvement was another critical feature of the software process. One of the guiding principles of the Shuttle software development was if a mistake was found, you should not just fix the mistake but must also fix whatever permitted the mistake in the first place. The process that followed the identification of any software error was: fix the error; identify the root cause of the fault; eliminate the process deficiency that let the fault be introduced and not detected earlier; and analyze the rest of the software for other, similar faults.

The goal was not to blame people for mistakes but instead to blame the process. The development process was a team effort; no one person was ever solely responsible for writing or inspecting the code. Thus there was accountability, but accountability was assigned to the group as a whole.

Carefully Defined Communication Channels. Such a large project and its long-term nature created communication problems. In response to the communication and coordination problems during Apollo development, NASA had created a control board structure, which was extended for the Shuttle. Membership on the review boards included representatives from all affected project areas, which enhanced communication among functional organizations and provided a mechanism to achieve strict configuration control. Changes to approved configuration baselines, which resulted from design changes, requirements change requests, and discrepancy reports, were coordinated through the appropriate boards and ultimately approved by NASA. Audits to verify consistency between approved baselines and reported baselines were performed weekly by the project office. In addition, the review checkpoints, occurring at critical times in development, that had been created for Apollo were again used and expanded.

Testing and Code Reviews. A final important feature of the development process with respect to achieving high quality involved extensive testing and code reviews: Emphasis was placed on early error detection, starting with requirements. Extensive developer and verifier code reviews in a moderated environment were used. It is now widely recognized that human code reviews are a highly effective way to detect errors in software, and they appear to have been very effective in this environment too.

A Professional Software Development Culture

Culture matters. There was a strong sense of camaraderie and a feeling that what they were doing was important. Many of the software developers worked on the project for a long time, sometimes their whole career. They knew the astronauts, many of whom were their personal friends and neighbors. These factors led to a culture that was quality focused and believed in zero defects.

The Shuttle software development job entailed regular 8 A.M. to 5 P.M. hours, where late nights were an exception. The atmosphere and the people were very professional and of the highest caliber. Words that have been used to describe them include businesslike, orderly, detail-oriented, and methodical. Smith and Cusumano note they produced "grownup software and the way they do it is by being grown-ups."[6]

The culture was intolerant of "ego-driven hotshots": "In the Shuttle's culture, there are no superstar programmers. The whole approach to developing software is intentionally designed not to rely on any particular person."[6] The cowboy culture that flourishes in some software development companies today was discouraged.

The culture was also intolerant of creativity with respect to individual coding styles. People were encouraged to channel their creativity into

Calendar of Events

VOLUTPAT ORNARE ARCU
Donec sit amet neque nec odio pharetra semper. Suspendisse dictum ligula eu diam. Pellentesque convallis porttitor eros. Nunc placerat accumsan ante. Etiam scelerisque nisl non ligula. Quisque vitae lacus. Pellentesque in augue. Integer laoreet nisl nec ipsum. Ut massa orci molestie quis, blandit et cursus et lorem. Donec congue massa quis metus.

DONEC EU MAGNA
Nunc aliquet ante eget lectus. Vestibulum scelerisque dignissim nisi. Phasellus id elit suspendisse aliquet. Aenean semper, magna quis interdum sagittis, arcu odio tincidunt lacus, non tristique diam arcu sed nibh. Vestibulum non eros vitae dolor dignissim volutpat.

Suspendisse elementum, felis vel hendrerit congue, neque urna consectetuer nisl, ac vehicula nisi leo id arcu. Aenean aliquam. Sed suscipit. Quisque semper justo sed leo. Aenean porta, diam non pellentesque pulvinar, ipsum orci ultrices dui, in elementum velit mauris sit amet dolor.

PELLENTESQUE ERAT
Vitae dui semper fermentum. Fusce pede mauris, rutrum at, ullamcorper porta, ultrices ac, felis. Integer nunc enim, bibendum quis, ullamcorper nec, dictum sed, lorem. Morbi lacinia felis vitae massa. Nam tortor magna posuere, adipiscing ac, tincidunt eu, lectus. Nulla tortor nisi, sodales non, luctus non, posuere at, ante. Suspendisse adipiscing sem mollis mi. Duis lobortis commodo orci.

ODIO SED TORTOR
Interdum mollis. Maecenas lobortis, tellus sed mollis nonummy, sapien ante aliquet tellus, et sagittis lacus dolor eu sem. Quisque ut turpis nec risus molestie scelerisque. Nulla placerat. Curabitur sollicitudin quam ut risus.

Mauris aliquet, felis imperdiet adipiscing imperdiet, purus dolor sollicitudin felis, vel convallis ligula lorem scelerisque lorem. Nunc pellentesque. Cras nec lacus. Aenean suscipit sem.

improving the process, not violating strict coding standards. In the few occasions when the standards were violated, resulting in an error in the flight software, they relearned the fallacy of waiving standards for small short-term savings in implementation time, code space, or computer performance.

A larger number of women were involved in the Shuttle software engineering than is common in the software development world today. Many of these women were senior managers or senior technical staff. Smith and Cusumano[6] suggest the stability and professionalism may have been particularly appealing to women.

The challenging work, cooperative environment, and enjoyable working conditions encouraged people to stay with the Shuttle software project. As those experts passed on their knowledge, they established a culture of quality and cooperation that persisted throughout the program and the decades of Shuttle operations and software maintenance activities.

Limitations of the Process

No development process is perfect and the Shuttle software is no exception. Various external reviews identified gaps in the process that needed to be filled. One was that the verification and validation inspections by developers did not pay enough attention to off-nominal cases.

A second deficiency identified was a lack of system safety focus by the software developers and limited interactions with system safety engineering. System-level hazards were not traced to the software requirements, components, or functions.

A final identified weakness related to system engineering. The NRC committee studying Shuttle safety after the Challenger accident recommended that NASA implement better top-down, system engineering analysis, including system safety analysis.[7]

Conclusion

The Shuttle software was not perfect, although it was better than most software today. Errors occurred in flight or were found in software that had flown. None of these software errors led to the loss of the Shuttle, although some almost led to the loss of expensive hardware and some did lead to not fully achieving mission objectives, at least by using the software: Because the orbital functions of the Shuttle software were not fully autonomous, astronauts or Mission Control could usually step in and manually recover from the few software problems that did occur. This too is a major lesson that should be learned by those rushing to make totally autonomous systems today.

Software engineering seems to be moving in the opposite direction from the process used for the Shuttle software development.

The few errors in the flight software should not detract from the excellent processes used for the Shuttle software development. When errors were found, they were usually traced to temporary lapses in the rigorous processes or to periods of lowered morale. One takeaway is that there is more to achieving high quality than simply rigorous processes (as promoted by Taylorists in the guise of such process-heavy concepts as CMM and CMMI). The culture of the development environment may be just as important or maybe more so.

Beyond the lessons learned that have been noted so far, some general conclusions can be drawn from the Shuttle experience to provide guidance for the future. One is that high-quality software is possible but requires a desire to do so and an investment of time and resources. Software quality is often given lip service in many industries, where frequently speed and cost are the major factors considered, quality simply needs to be "good enough," and frequent corrective updates are the norm.

Software engineering seems to be moving in the opposite direction from the process used for the Shuttle software development, with requirements and careful pre-planning relegated to a less important position than starting to code. Strangely, in many cases, a requirements specification is seen as something that is generated after the software design is complete or at least after coding has started. Why has it been so difficult for software engineering to adopt the disciplined practices of the other engineering fields? There are still many software development projects that depend on cowboy programmers and "heroism" and less than professional engineering environments.

Ironically, many of the factors that led to success in the Shuttle software were related to limitations of computer hardware in that era, including limitations in memory that prevented today's common "requirements creep" and uncontrolled growth in functionality as well as requiring careful functional decomposition of the system requirements in order to break it into small pieces that could be loaded only when needed. Without these physical limitations that impose discipline on the development process, we need to determine how to impose discipline on ourselves and today's safety-critical projects. C

References

1. Broache, A. IRS trudges on with aging computers. CNET News (Apr. 12, 2007); http://news.cnet.com/2100-1028_3-6175657.html.
2. Eggan, D. and Witte, G. The FBI's upgrade that wasn't. *The Washington Post* (Aug. 18, 2006); http://www.washingtonpost.com/wp-dyn/content/article/2006/08/17/AR2006081701485.html.
3. Johnson, K. Denver airport saw the future. It didn't work. *New York Times*, (Aug. 27, 2005); http://www.nytimes.com/2005/08/27/national/27denver.html?pagewanted=all&_r=0.
4. Lewyn, M. Flying in place: The FAA's air control fiasco. *Business Week* (Apr. 26, 1993), 87, 90.
5. *Post-Challenger Evaluation of Space Shuttle Risk Assessment and Management.* Aeronautics and Space Engineering Board, National Research Council, January 1988.
6. Smith, S.A. and Cusumano, M.A. *Beyond the Software Factory: A Comparison of Classic and PC Software Developers.* Massachusetts Institute of Technology, Sloan School WP#3607=93\BPS, (Sept. 1, 1993).
7. Tomayko, J. *Computers in Spaceflight: The NASA Experience.* NASA Contractor Report CR-182505, 1988.

Nancy G. Leveson (leveson@mit.edu) is Professor of Aeronautics and Astronautics and also Professor of Engineering Systems at Massachusetts Institute of Technology, Cambridge, MA.

This column has been derived from the chapter "Software and the Challenge of Flight Control" of the book *Space Shuttle Legacy: How We Did It/What We Learned*, R. Launius, J. Craig, and J. Krige, Eds., 2013. The complete chapter can be downloaded from http://sunnyday.mit.edu/STAMP-publications.html.