2009

Flight Software Engineering Lessons

Ronald Kirk Kandt
California Institute of Technology, ronald.k.kandt@jpl.nasa.gov

Flight Software Engineering Lessons

Ronald Kirk Kandt
Jet Propulsion Laboratory, California Institute of Technology
ronald.k.kandt@jpl.nasa.gov

ABSTRACT

This paper summarizes the findings of several studies and mishap investigations that explored the causes of flight software problems. From these studies and mishap investigations the author identifies several lessons from which others can learn. A key lesson that the author proposes is that software engineering in the real world is largely one of managing risk.

Keywords

Software engineering, software process improvement.

INTRODUCTION

Jet Propulsion Laboratory (JPL) is a Federally Funded Research and Development Center managed by the California Institute of Technology under contract to the National Aeronautics and Space Administration (NASA). JPL was formally founded by a small group of men interested in rocketry in 1943 (Pendle, 2006). Today, the primary mission of JPL is to explore and observe the farthest reaches of the solar system (Westwick, 2006). Some of the more recent missions include Cassini-Huygens (Harland, 2002; Lorenz and Mitton, 2002), Mars Pathfinder (Muirhead and Simon, 1999; Shirley, 1999), Mars Exploration Rovers (Squyres, 2006), and Spitzer Space Telescope (Rieke, 2006).

All of JPL's missions are extremely challenging. For example, consider the missions to Mars through 1999 (Godwin, 2000). JPL was responsible for 13 missions and had 7 successes. Other entities had 17 missions to Mars, all of which failed. Since 1999, JPL has had 5 more successful missions to Mars and no failures (See http://mars.jpl.nasa.gov/). In sum, space exploration is a risky business, but JPL is learning how to do it better.

There are several reasons for failed space missions. Many of the recent failures, as well as significant problems nearly leading to failure, have been attributed to software (JPL Special Review Board, 1999; JPL Special Review Board, 2000; NASA, 1999). The characteristics of the software and the way it is developed often lead to failure.

The motivation for this work is to identify lessons that will help to prevent the failure of future space exploration missions. Currently, there is a significant amount of knowledge about how to develop general-purpose software. However, there is little documented knowledge of how to perform hardware and software co-development and almost no publicly accessible information is available that discusses the development of flight software. The contribution of this work is one effort to increase this knowledge. The goals of this work are two-fold. First, it attempts to identify what has not worked well in the past. Second, it proposes practices that will reduce the number of software failures in the future.

THE SOFTWARE ENGINEERING CONTEXT

The engineering of flight software is tightly coupled with the engineering of the spacecraft, the engineering of the instruments that comprise the spacecraft's payload, and the engineering of software that simulates the spacecraft, its payload, and the operational environment of space. All of these activities are performed concurrently, which makes it extremely challenging to develop verified flight software.

Spacecraft are composed of many mechanical and electronic elements, but the ones most affecting the development of flight software are the computational elements – flight processors, busses, memory, interface controllers, etc. – that reside on several custom cards that are mounted in a custom chassis. Common-off-the-shelf flight processors and memory devices that operate in the extreme environments of space (e.g., radiation, temperatures below -100° C) are used by JPL in its spacecraft. However, JPL makes the remaining computing elements. Because of this, initial versions of the computing elements that JPL makes often do not provide the full functionality that they are suppose to provide and often have design defects that are found during hardware testing or during use by flight software personnel. In addition, only a small number of each element is manufactured and the demand for their use exceeds their availability. Like these elements, the development of instruments – cameras, radiometers, and so on – follow largely the same process and share the same issues.

As hardware elements are being developed, software engineers writing the simulation software need access to design documentation. Unfortunately, the design documentation often is either not available or is inaccurate. In this environment, these software engineers are expected to produce software simulations of the spacecraft and its payload. The consequence of this is that early versions of the simulation software seldom accurately reflect the behavior of the spacecraft and its payload and reasonably accurate simulations are often not available until close to the end of the development lifecycle.

In spite of this, flight software engineers develop flight software and test it in many ways. One way is to test it against older spacecraft or older spacecraft simulations until new simulations of the actual spacecraft and its payload are available. Once these new simulations are available, flight software engineers will test software against them. However, the initial simulations tend to be accurate only in a logical sense, whereas the later simulations tend to be more accurate with respect to timing issues. The consequence of this is that flight software engineers can only test limited aspects of their software until they gain access to flight-like hardware, which generally occurs late in the development lifecycle.

Once flight software engineers gain access to flight-like testbeds, they are able to test their software against hardware that reflects the actual behaviors and performance characteristics of the hardware that will operate in space. At this point, some of the tests that previously succeeded in the simulation environments will fail. Because flight software engineers are given so little time to use the flight-like testbeds before launch, flight software engineers perform the bulk of their testing after the spacecraft has been launched and then upload software changes before critical events, like landing on Mars. This generally gives flight software engineers 6-9 months of test time during flight.

RESEARCH METHODOLOGY

The method that the author used to perform this work was to summarize the findings of 3 studies done at JPL and 3 mishap investigations done by review boards comprised of people from JPL, NASA, and industry. The author was the lead author of one of the JPL studies. After summarizing the findings, the author proposes practices, stated as lessons, to reduce the likelihood of failures reoccurring in the future. The key activities performed by the members of each study and mishap investigation are described below.

Studies

Three studies were performed that form a basis of the research described herein (Hihn and Habib-agahi, 1991; Hihn and Habib-agahi, 2000; Kandt, Kay-Im, Lavin, and Wax, 2002). The people that performed these studies had expert knowledge of the topics of each study. For each study, these people performed the following activities.

Identify a representative sample of the general population to participate in the study. This meant that the people that were interviewed performed the tasks concerning the underlying studies, fairly represented the various sections that develop software at JPL, and fairly represented the experience of personnel that generally performed the tasks for which they were being asked questions. One of these studies stated that the distribution of the sample population and the actual workforce had less than a 1% probability of being different (Hihn and Habib-agahi, 1991).

Interview each study participant. In general, each interview began with collecting personal information, such as the years of experience developing software and the years of experience performing specific software tasks. Then, the interviewers generally asked each participant a pre-defined structured set of questions relevant to the goals of the study. For example, to identify issues related to software technology use, participants were asked questions related to the use of specific software methodologies, techniques, and tools. Each question was written to avoid biasing the interviewee's response. Although there was a structured set of questions, we were flexible during each interview so that we could discuss unanticipated issues. Interviewers concluded each interview by asking participants to describe, in their own words, what they did, how they did it, and what worked and did not work.

However, some interviews were performed in an unstructured manner using protocol analysis (Ericsson and Simon, 1993). The purposes of these interviews generally were to verify the conclusions being made by the researchers. These interviews often began with the researchers describing their understanding of how people generally did their work, including variations to the generalizations. Once the researcher finished with the description, interviewees tended to describe what they did differently from the proposed description and why they did things differently. This usually led to a free-flowing exchange of information that lasted one to two hours, which was later analyzed for nuggets of wisdom.

Summarize each interview. The researchers generated a transcript of each interview and summarized each interview. The participant of each interview was then given the transcript and summarization for his or her interview. All inaccuracies and clarifications provided by the participants were then used to revise the interview transcripts and summaries.

Analyze the interview results. The primary strategy used in these surveys was to use binary and categorical variables to describe the data that they collected. The researchers analyzed this data using various statistical means. In addition, researchers generated qualitative results from anecdotal comments that were not readily categorized.

Generate a report. Each study resulted in the researchers generating a report that identified the objectives of the study, the topics that were studied, the characteristics of the study participants, the questions asked of the study participants, the transcripts of all interviews, the analyses performed, and the conclusions that the researchers made. Afterwards, the report is provided to management.

Mishap Investigations

Three mishap investigations were performed that form a basis for the research described herein (JPL Special Review Board, 1999; JPL Special Review Board, 2000; NASA, 1999). Each of these investigations performed the following activities.

Identify the sequence of events leading up to a failure. The investigation board collects the command sequences that had been uploaded to the spacecraft and the telemetry data that had been sent to the ground before the mission failure. After collecting this data, it is analyzed to identify any operational procedures that deviated from the norm or violated specified flight rules. Flight rules constrain how operations personal operate a spacecraft.

Interview key project personnel. The investigation board interviews key personnel – project manager, systems engineers, flight operators – to determine anomalies or deviances from planned operations that had occurred over the recent past, whether formally reported or not. Often, the operational context at the time of mission failure sheds light onto what could go wrong.

Identify the potential causes of the failure. To perform this task, the investigation board uses the information collected from event sequences, telemetry data, and interviews, as well as spacecraft design information. Especially important is the identification of each existing Failure Mode, Effects, and Criticality Analysis and Fault Tree Analysis that project personnel performed. From the potential causes of failure, the investigation board will identify a most likely cause of failure.

Generate a report. The investigation board summarizes the results of the activities that it performed as a report. This report identifies the failure that occurred, the sequence of events leading to the failure, several likely causes of the failure, and the most likely cause of the failure. In addition, the report makes several recommendations to prevent the failure from occurring in future missions. This report is then widely disseminated within NASA and its suppliers.

FINDINGS

Over the past two decades there have been several accounts that identify problems that JPL personnel have encountered while developing flight missions; the 6 surveys and mishap investigations identified earlier are summarized below. The importance of these accounts is that despite changes in technologies and methodologies, the core set of significant problems have remained constant. This implies that we need to better understand why these problems continue to reoccur and then attempt to eliminate these problems using a different approach. In this paper, this information is captured as learned lessons.

Studies

In 1991, a study was done that attempted to identify how software cost estimates were done at JPL (Hihn and Habib-agahi, 1991). In this study, 83 people completed a questionnaire on software cost estimation practices and 48 people participated in a software size and effort estimation experiment. This study resulted in the following key findings:

- Every mission requires a significant amount of new software to implement functionality that has never been provided before.
- Contractual environments often provide incentives for an organization to underbid a proposal so that it can win a contract.

The impact of these two findings is that there is a 30% probability that a software cost estimate is incorrect by more than 50% and a 67% likelihood that a cost estimate will be off by more than 20%.

In 1999, a follow-up study to the 1991 study was done (Hihn and Habib-agahi, 2000). In this study, 11 managers and lead software engineers were interviewed. The study resulted in several key findings related to software:

- Software reuse estimates were overly optimistic; reuse estimates were incorrect by 25%.
- Software staff did not assist in the allocation of requirements to software.
- Requirements changed throughout the software development lifecycle.

- There was insufficient traceability between requirements.
- Software architectures were often inadequate.
- Software simulators were not available until late in the software development lifecycle.
- There were an insufficient number of testbeds available for use by software engineers and these testbeds had limited functional capability.

As a result of these practices, software development costs exceeded software development allocations by 50%, on average.

In 2001, a study was performed to determine the software development processes that were used at JPL, as well as to identify software tools that people used to develop software (Kandt et al., 2002). In this study, 20 managers and lead software engineers were interviewed. Key process findings of this study were:

- Software delivery dates are defined by launch opportunities, with little room for adjustment.
- An initial set of software requirements is difficult to obtain, which delays design.
- Most people had difficulty using the requirements engineering tool that the institution provides.
- There is limited reuse of design and code; often high-level design is inherited, but detailed design and code are often done from scratch.
- Development is delayed because the development and test environments are not ready when needed, which is often caused by hardware design lags.

Mishap Investigations

After the Mars Climate Orbiter (MCO) spacecraft was lost, two investigations were performed to diagnose the root cause of the mission failure (JPL Special Review Board, 1999; NASA, 1999). It was determined that the root cause was the failure to use metric units in a ground software file, which was generated by a program. Specifically, thruster performance data was given in English units instead of metric units. This occurred even though the software interface specification specified the use of metric units. This problem was not noticed earlier because this file was not used during operations because it had multiple format errors and incorrect numeric specifications. Instead, operations personnel used an alternative strategy for producing the right data. Four months later, the file was corrected and within a week operations personnel observed that the file contained anomalous data. Key findings of these studies were:

- Rigorous criteria were not used to identify mission critical software.
- Several reviews occurred without the attendance of key personnel. For example, operations navigation personnel did not attend Preliminary Design Reviews (PDRs) or Critical Design Reviews (CDRs).[1]
- The verification and validation process did not adequately address ground software. Specifically, end-to-end testing was not done.[2] In addition, verification of ground system interfaces was either haphazardly performed or not completed.
- Problems were dealt with informally instead of using a formal Incident, Surprise, and Anomaly (ISA) process.[3]

Similarly, after the loss of the Mars Polar Lander (MPL) mission, another investigation was performed to determine the cause of its loss (JPL Special Review Board, 2000). Unfortunately, this was difficult to do in the absence of telemetry data. However, it was determined that the most probable cause of the MPL mission failure was the premature shutdown of the descent engines, caused by the vulnerability of the software to transient signals. The key findings of this study were:

---

[1] The purpose of a PDR is to assure that the proposed architectural design and associated implementation approach will satisfy the system and subsystem functional requirements. The purpose of a CDR is to assure that the detailed technical design is correct and satisfies all allocated requirements.

[2] End-to-end testing of a spacecraft attempts, as much as possible, to "fly" the spacecraft while it is on the ground. Hence, the spacecraft is fully assembled on the ground, commands are sent to it while it is on the ground, and then the observed behavior is compared to what is expected. Of course, all commands cannot be sent to the spacecraft because some command sequences could harm the spacecraft while it is on the ground.

[3] An ISA report documents an unexpected observation once a spacecraft is operational. An ISA process identifies how to report such anomalies and take corrective actions.

- The MPL mission suffered from severe and unprecedented technical and fiscal constraints, which were clearly unrealistic.
- Because of cost constraints, analysis and modeling was accepted as a lower-cost approach to inspection and test. Similarly, cost constraints caused technical issues raised at PDRs and CDRs to be closed without independent technical review. It was found that although the appropriate concerns were raised at PDRs and CDRs, the closure actions were sometimes inadequate.
- Software system design activities lagged behind the design of other subsystems and the potential failure modes were not adequately addressed. Specific design weaknesses were found in the initialization of software.
- The flight software was inadequately tested and it was not tested in a flight configuration. For example, adequate testing in a flight configuration would have discovered several problems associated with post-landing fault-response algorithms.
- Many people performed multiple project roles, which had the adverse effect of eliminating the necessary checks and balances normally found in projects.

LESSONS LEARNED

These studies and mishap investigations involved numerous missions and people over two decades. Even so, several problems reoccurred regardless of the level of software maturity of the engineering organization performing the work. For example, JPL recently achieved CMMI maturity level 3, yet a recent project that was part of the CMMI appraisal still suffered from some of the problems identified by these studies and investigations. When analyzing the environment that JPL must work in, I have learned the following valuable lessons.

Lesson 1: Adopt a risk-based approach to software engineering. The reasons for doing this are:

- Funds for developing flight software will almost certainly be inadequate because (a) software reuse estimates are almost always overly optimistic, (b) every mission requires a significant amount of new software that is difficult to accurately cost, (c) system responsibilities originally allocated to hardware will be fulfilled by software, (d) hardware will operate differently than originally planned, and (e) contracts are often bid using extremely optimistic cost assumptions.
- The length of the software development lifecycle will be reduced because hardware delivery dates will slip and flight software delivery dates are firm, due to launch opportunities.
- Flight-like testbeds will almost certainly be delivered late, which means rigorous testing of software will have to be delayed until engineering models of the hardware are available.

Furthermore, the reasons for selecting a risk-based approach to software engineering also implies that CMMI is not an appropriate model for the types of missions that JPL undertakes. For example, *Generic Practice 2.3 – Provide adequate resources for performing the process, developing the work products, and providing the services of the process* is almost always violated. That is, seldom is enough funds provided to execute every CMMI practice in a manner that yield benefits. Instead, JPL flight software teams must evaluate the benefits, costs, and risks of performing or not performing each practice while considering the funding that's available to it. The goal of risk-based software engineering should be to perform those practices having the greatest value, while achieving an acceptable level of risk. Adopting each of the following lessons minimizes software engineering risk.

Lesson 2: Software engineers must be involved in early system-level decisions that determine the role that software will play in the performance of system functions. Their participation will ensure that fully informed decisions are made regarding cost and desired system qualities (e.g., performance, reliability, flexibility).

Lesson 3: The software development infrastructure must be available before staffing a project with software engineers. The reason for this is that once a project is staffed with software engineers, the project will not have sufficient time to correct any inadequacies with individual tools or their integration. In addition, changes to the software development process once software engineers have been assigned to a project have the potential to disrupt progress.

Lesson 4: Develop simulations of all the instruments and other custom hardware that interfaces with the flight software as early as possible (e.g., whenever a relatively stable document exists that describes the behaviors of an instrument or other custom device). These simulations will permit the development of regression test suites that test the logic of flight software, which can later be reused for testing software in the flight-like environment. This approach avoids the *big bang* approach to testing at the end of the lifecycle, which is almost always inadequate.

Lesson 5: Develop flight software architectures before coding begins and use these architectures to estimate the performance characteristics of the integrated flight software and hardware system. Later, as code is developed, substitute estimates of

resource usage and performance characteristics based on actual data. Then, use the differences between estimates and actual data to reassess the architecture and the detailed technical decisions already made.

Lesson 6: Develop reference architectures that integrate both hardware and software. Each reference architecture must be able to support a common mix of payload characteristics of current and future flight systems. The development of such architectures will significantly ease problems associated with the instability of requirements and hardware and unavailability of software simulations and flight-like testbeds.

Lesson 7: Use a robust method for engineering requirements. Such a method should require:

- The definition of performance and resource utilization requirements,
- The tracing of requirements to higher- and lower-level requirements,
- The review of requirements by key stakeholders, including those not directly involved in the engineering of those requirements, and
- The assessment of requirements using a checklist of questions that address quality concerns.

This will help an organization develop a complete and consistent set of requirements.

Lesson 8: Use objective measures for monitoring development progress and determining the adequacy of software verification activities. These measures should identify:

- The percentage of code tested,
- The percentage of requirements tested,
- The percentage of defined faults tested,
- The percentage of tests passed in a simulation environment,
- The percentage of tests passed in a testbed environment,
- The number of units[4] where the allocated requirements have been baselined,
- The number of units where the detailed design has been baselined,
- The number of units where coding has been completed and successfully passed all unit tests in a simulation environment,
- The number of units where coding has been completed and successfully passed all unit tests in a testbed environment,
- The number of units that have successfully passed all stress tests, and
- The percentage of code of the delivered units that was actually reused.

Without collecting these measures, one will not be able to reliably assess the progress that is being made by software engineers or the quality of products that they produce.

Lesson 9: Use an integrated system for managing work. At JPL, we use various action item tracking systems, problem reporting systems, and scheduling systems. Unfortunately, none of them are integrated, which prevents managers from appropriately monitoring work. For example, one project uses:

- An institutionally developed system to manage action items, which uses one database system,
- An institutionally developed problem reporting system to manage anomaly reports, which uses a database system different from the one used by the action item tracking system,
- A commercial product for scheduling activities performed by software engineers, and
- A commercial product for defining requirements, which are not linked to the capabilities for scheduling work that is assigned to software engineers.

Since all of these systems are stovepipes, a lot of work is not accounted by the scheduling system, which is used to determine if a project is on track. For example, if a person spent 20 hours correcting a problem or performing the activity described by

[4] A unit, in this context, is meant to be the lowest level of the decomposition of a software architecture. In the literature, these are sometimes called components or modules. A unit is generally composed as 3,000 to 5,000 lines of source code.

an action item, the schedule would not reflect that 20 hours of work was done and it would not reflect that the person performing this work was now one-half week behind schedule.

CONCLUSIONS

This paper has identified numerous problems that continually occur at JPL and pose risk to its future projects. Many of these problems are familiar to most readers (e.g., inadequate funding, deficiencies in requirements, and inadequate testing tools) and are some of the leading causes of software problems (Jones, 1994; Kandt, 2006; Neufelder, 1992). Root causes of these problems are insufficient funding and inadequate development schedules. Unfortunately, JPL rarely can change this situation and must deal with it as best it can.

The consequence of these realities is that almost every mission that JPL participates on involves at least moderate risk. Consequently, JPL must focus its effort in making rationale decisions based on the information available to it. To make rationale decisions, JPL must analyze key trades and choose those that have minimal, or at least acceptable, risks. Part of the process of making rationale decisions is addressing past failures and other known risks and adopting strategies that mitigate those risks. This is what this paper has proposed that future JPL missions should do.

Because the problems that are identified in this paper frequently occur in most software development efforts, the lessons identified in this paper are applicable to other software development organizations too. For example, many projects suffer from inadequate funding and tight schedule constraints just like JPL. Further, software tools for software professionals seldom work as well as advertised and rarely work well with one another. This is an industry problem that has existed since the development of the first computer.

ACKNOWLEDGMENTS

Milt Lavin and Timo Käkölä reviewed preliminary versions of this paper; their comments helped to improve its quality. The suggestions of an anonymous reviewer also were very helpful. This work was performed at the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration.

REFERENCES

1. Ericsson, K. A. and Simon, H. A. (1993) Protocol Analysis: Verbal Reports as Data, The MIT Press.
2. Godwin, R. (2000) Mars: The NASA Mission Reports, Volume 1, Apogee Books.
3. Harland, D. M. (2002) Mission to Saturn: Cassini and the Huygens Probe, Springer-Verlag.
4. Hihn, J. and Habib-agahi, H. (1991) Cost Estimation of Software Intensive Projects: A Survey of Current Practices, *Proceedings of the International Conference on Software Engineering*, 276-287.
5. Hihn, J. and Habib-agahi, H. (2000) Flight Software Cost Growth: Causes and Recommendations, Jet Propulsion Laboratory Document 18660.
6. Jones, C. (1994) Assessment and Control of Software Risks, Prentice Hall.
7. JPL Special Review Board (1999) Report on the Loss of the Mars Climate Orbiter Mission, Jet Propulsion Laboratory Document 18441.
8. JPL Special Review Board (2000) Report on the Loss of the Mars Polar Lander and Deep Space 2 Missions, Jet Propulsion Laboratory Document 18709.
9. Kandt, R. K., Kay-Im, E., Lavin, M. L. and Wax, A. (2002) A Survey of Software Tools and Practices in Use at Jet Propulsion Laboratory, Jet Propulsion Laboratory Document 24868.
10. Kandt, R. K. (2006) Software Engineering Quality Practices, Auerbach.
11. Kandt, R. K. (2009) Experiences in Improving Flight Software Development Practices, *IEEE Software*, May/June.
12. Lorenz, R. and Mitton, J. (2002) Lifting Titan's Veil: Exploring the Giant Moon of Saturn, Cambridge University Press.
13. Muirhead, B. K. and Simon, W. L. (1999) High Velocity Leadership: The Mars Pathfinder Approach to Faster, Better, Cheaper, HarperCollins Publishers.
14. NASA (1999) Mars Climate Orbiter Mishap Investigation Board Phase I Report.
15. Neufelder, A. M. (1992) Ensuring Software Reliability, Marcel Dekker.
16. Pendle, G. (2006) Strange Angel: The Otherworldly Life of Rocket Scientist John Whiteside Parsons, Harvest Books.

17. Rieke, G. H. (2006) The Last of the Great Observatories: Spitzer and the Era of Faster, Better, Cheaper at NASA, University of Arizona Press.
18. Shirley, D. (1999) Managing Martians, Broadway.
19. Squyres, S. (2006) Roving Mars: Spirit, Opportunity, and the Exploration of the Red Planet, Hyperion.
20. Westwick, P. J. (2006) Into the Black: JPL and the American Space Program, 1976-2004, Yale University Press.