Breaking The Cause And Effect Cycle

Or

Lessons Discovered But Seldom Learned

Or

Why Am I Doing This If No One Listens?

Dave Hall
Principal Engineer
MDA/QS
October 2006

Since engineering started mistakes, errors, arrogance and lack of knowledge have plagued the successful accomplishment of all programs. Looking over the last four decades, program developers, program managers, senior management, risk management practitioners and other forward-thinking types have prepared "Lessons Learned" on individual programs. Such "Lessons Learned" reports (or risk identification checklists) have been accomplished on all types of government, industrial and commercial programs, operations and activities and just about everything else. Unfortunately, little use has been made of this valuable data source other than through the experience of personnel who have moved from one program to another. There are some shining examples but, on the whole, very few people seem interested in learning from "Lessons Learned" (or failure data). The *Effects* of these *Causes* are program failures. It is obvious from the repetitive nature of these causes that most programs do not take advantage of "Lessons Learned" from prior programs. Based on the statistics of successful versus unsuccessful programs over the past four decades, I believe that if we could determine an effective and efficient way to understand ***and use*** Lessons Learned, we could significantly increase the probability of having a successful program.

The Problem

The problem as I see it is that it is easier to identify the cause of a problem than to get a person or an organization to act on that knowledge and implement a useful solution or be concerned about the exact same thing happening in a subsequent program. For that reason, I like to use the title "Lessons Discovered" to make clear that just noting the cause of a problem does not solve it or require one to consider it for future programs. When you identify a problem cause, you are calling into question the wisdom of earlier decisions made by yourself, other people or your management. Most managers and organizations do not take kindly to "criticism". So it is unlikely that the real "Lesson Learned" will be spelled out in sufficient detail to be effective (how many people will admit to the details of their wrong decision?). Even if an organization gets the development of Lessons Learned mandated for their programs; excuses and creative explanations will emerge if a lesson learned threatens some cherished program or mode of operation. And to consider that one might make the same "bad decisions" in the future is simply out of the question.

Another major stumbling block to actual use of lessons learned is that managers and developers are very seldom (if ever) held responsible for unsuccessful programs. I am not advocating

holding someone responsible for simply failing – failure happens and provides people with experience. I am advocating being held responsible for failures that occur from the same root cause(s) every time. This indicates to me that management and/or developers and/or operators are refusing to learn from past mistakes – their "Lessons Not Learned". Change of one's opinion or method of operation is difficult – why do it if nothing bad happens to me when I fail?

And there are other problems as well. Lessons Learned often become twisted to support - or at least not impact - pet projects. What very few people want is a totally dispassionate look at the lessons discovered. No one wants the chips to fall where they may. Too much collateral damage that way. Yet, in the end, truth and logic will have their way. The true meaning of each lesson learned will be there as a cause of problems on the next program or operation or activity, whether you have come up with the best implementation of the lesson or not. But does anyone care?

Examples of Lessons Learned But Not Used

Depending upon the unique aspects of a program, there have always been a multitude of reasons for being unsuccessful. These reasons have usually included the following:

- **Sloppy requirements:** Every program depends upon solid user requirements being firmly locked down prior to any work being undertaken. Failure to do so is a leading cause for program failure. Somehow, the trend is that many program teams think they can get started by rushing the requirements-gathering phase. These programs are then eagerly started with incomplete requirements. And they are unsuccessful. Do you know any programs that were unsuccessful due to incomplete requirements? Why didn't those programs define their requirements?
- **Scope creep:** When customers insist on ever-increasing changes to the product being developed, scope creep can jeopardize the program. An even more difficult situation for a program surfaces when new changes are introduced after the program has been launched. This usually drives up the cost, resource requirements, deliverables, and completion time. Do you know any programs that were unsuccessful due to scope creep? Why did those programs allow scope creep?
- **Poor planning and estimation:** Those programs that are poorly estimated and planned tend to fail both in cost and schedule, which eventually causes the overall program to fail. Managers tend to start programs without relying on proper analysis, and sizing, and fail to consult subject matter experts or cost estimators to validate how much program work packages will cost. Original schedules and critical paths are usually (nearly always) overly optimistic. *Everything will happen just as planned and there will be no need to deviate from this planned schedule.* However, once ongoing, program schedules spiral out of control when dates and deliverables aren't aggressively monitored and tracked on a daily basis. All too often, managers leave issues unresolved for days/weeks/months since either they can't make a decision or are micromanaging and no one else dares make a decision, which then results in schedule overruns. Original budgets are usually (nearly always) overly optimistic. *We must provide this low-ball estimate or the customer will not choose us. Once we get the contract, we assume that the customer will change the requirements and we can get well on budget. Everything will happen just as planned and there will be no need to deviate from the planned schedule and resource use.* Planning for future problems, allowing for risks to become problems, letting customers know about

the risks to successful completion of this program so you can maintain a management reserve is simply out of the question. Do you know any programs that were unsuccessful due to poor planning or inaccurate cost/schedule estimates? Why did those programs use poor estimates?

- **Poor documentation:** Maintaining inadequate program documentation is cause for concern and has raised the red flag innumerable times. Lessons discovered from many failed programs reveal that there was too little documentation to adequately describe the program even in broad terms and to serve as a clear communication channel between customer and contractor. Do you know any programs that were unsuccessful due to poor documentation? Why did those programs not require adequate documentation?
- **New technology:** Programs that require integrating new tools and deploying new software/hardware are always more difficult, because usually only the vendor engineers clearly understand the limitations and functionality of the products. This results in delays in program schedule and could require weeks or months before the system/products are stable enough to be deployed. If a program uses new technology, there are numerous checklists, each one based on the type of technology. Do you know of any programs that were unsuccessful due to poor decisions made concerning new technology? Why didn't those programs check into the known problems with the technology?
- **Disciplined execution:** Some of the most significant drivers to program failure are management and culture related. Technological failings, while they exist, also have a strong management flavor, as they tend to cluster around failings in the systems engineering process. A recent Defense Science Board report states: "Too often, programs lacked well thought-out, disciplined program management and/or software development processes. … In general, the technical issues, although difficult at times, were not the determining factor. Disciplined execution was." [DSB2000]. There are numerous Lessons Discovered to show how this lack of disciplined execution manifests in programs. Some deficiencies are related to human nature. Self-interest leads people to primarily consider their tenure on a job, cleaning up problems left for them by their predecessors and often not considering long-term consequences of short-term decisions. There is also a tendency to try to place blame on other people or organizations: customers and program offices cannot hold to a set of requirements; contractors don't live up to their obligations; vendor's products don't live up to their performance and capability claims. It is obviously someone else's fault. This is all a case of lack of discipline[1]. Do you know of any programs that were unsuccessful due to poor discipline? Why is this lack allowed to continue once noticed?
- **Poor communications:** One of the biggest reasons why any program is unsuccessful is a lack of communication between customer and contractor, different design teams, program personnel and managers, program managers and senior management, etc. Many programs fail simply because no one understands what to do and receives no communication as to current progress or lack thereof. Do you know of any programs that were unsuccessful due to poor communications? Why did those programs allow poor communications?

[1] Real-Time Systems Engineering: Lessons Learned from Independent Technical Assessments, Theodore F. Marz, Daniel Plakosh, *June 2001*, CMU/SEI-2001-TN-004

- **Poor decision-making:** Decisions that aren't made at all and decisions that are delayed due to second-guessing are both major causes of program failures. Additionally, decisions that have responsibility for them passed so far up the management line that they end up being made based on faulty, incomplete or no information also are major failure causes. Do you know of any programs that were unsuccessful due to poor decision making? Why didn't senior management take action once poor decision making was uncovered?
- **Poor or inexperienced program management:** The person managing the program may not have the skills or experience to pull it off. Do you know of any programs that were unsuccessful due to poor or inexperienced management? Why did those programs have poor or inexperienced management and why didn't senior management aid them?
- **Poor testing:** A big cause of program failure is having either too little testing or testing too late in the process. We know that both testing and quality assurance need to be built into the program from the day the program is launched. How many times have we seen testing cancelled or given short shrift because of schedule or budget concerns? A 2002 study commissioned by the National Institute of Standards and Technology found software bugs cost the U.S. economy about $59.5 billion annually. The same study found that more than a third of that cost (about $22.2 billion) could be eliminated by improving testing. Is this Lesson Discovered completely unknown to most software managers and developers? Do you know of any programs that were unsuccessful due to poor or incomplete testing? Why was testing reduced?

The programs these Lessons Learned are from are spread over fifty+ years and a number of different organizational types. Why didn't the managers, engineers, operators or the customers use these Lessons Discovered to minimize the risk of having an unsuccessful program? Are any of these causes new? Is there any reason to not expect one or more of them to be a risk factor in any type of program? Table 1 at the end of this paper also provides some indication that the above lessons have been discovered and rediscovered, but seldom learned. There are thousands of unsuccessful programs one could add to this list. In very few cases was the root cause of the program failure a totally new and unknown problem. Most of the failures were aided by an unwillingness to learn from past programs.

In looking at a different area, a recent study conducted at the Swiss Federal Institute of Technology in Zurich analyzed 800 cases of structural failure in which 504 people were killed, 592 people injured, and millions of dollars of damage incurred. When engineers were at fault, the researchers classified the causes of failure as follows:

- Insufficient knowledge - 36%
- Underestimation of influence - 16%
- Ignorance, carelessness, negligence - 14%
- Forgetfulness, error - 13%
- Relying upon others without sufficient control - 9%
- Objectively unknown situation - 7%
- Imprecise definition of responsibilities - 1%
- Choice of bad quality - 1%
- Other - 3%

These cases are also spread over a number of years and a number of different structural types. Why didn't the engineers or the customers use Lessons Learned from earlier programs to minimize the risk of having an unsuccessful program – structural failure? Are any of these causes new? Is there any reason to not expect one or more of them to be a risk factor in a program of this type?

Conclusions

How can we break the Cause and Effect cycle? How can we get people to use Lessons Learned more effectively? Why don't more people and organizations actually use history, experience and knowledge to increase their program success rate? I have found that in programs that are in trouble, there are NO innocent parties. All stakeholders involved have participated (at some level) in creating or abetting the failure. And failure, in almost all organizations and areas, is common.

After working with a number of programs and many different people, several of us in the mission assurance area have made the following observations[2]:

1. Most people, engineers especially, are not trained to think of failing in their tasks. *Researching Lessons Learned is not useful since no such problems will occur on my watch.*
2. There is a bias against bad news. Some managers do not want to hear anything negative. Therefore they do not want to acknowledge that their program may have risks and/or problems. *Shooting the messenger immediately takes care of the problem.*
3. There is never enough time or funds to do this right. *However, there is always enough time and funds to clean up the failure.*
4. Managers don't want to know about problems because it is too painful. *Maybe if I ignore the problem it will go away on its own.*
5. There are individuals and organizations that like working in or with ambiguities. Therefore, they are never pined down to a solution or committed to an answer. *They are always able to change direction without being noticed, because there was never any direction to begin with.*

Are programs that you have worked with or are currently working with facing any of these problems? I expect so and also that you have already communicated them to the management. So was anything or has anything been done about them? And how are your Lessons Learned being preserved and passed along to the new programs so these problems could be avoided in the future? Have your Program Managers and senior leadership learned to change the decision patterns that cause problems in their new programs or are they continuing with their old habits? In most case, the answer to these questions is unfortunately the obvious one.

So how do we get people, programs and organizations to actually use Lessons Learned to break the Cause and Effect Cycle? How do we turn indifference and unwillingness to learn into something different? We need to convince managers and engineers and operators to go out of their way to seek and welcome evidence that seems to confute or contradict the experience and wisdom of their own most cherished beliefs. We need to establish a learning approach in our organziations. Such an approach would involve at least four steps:

---

[2] Some of these observations are paraphrased from Risk Management - Lessons Learned And Observations, by G. Jeffrey Robinette, INCOSE Systems Engineering Symposium, 2004

1. **Recognition**. Recognise that problems and failures exist, and that they provide opportunities for generating knowledge.
2. **Capture Knowledge**. Find ways to capture the knowledge generated by the program. Ways of doing this include regular review meetings during the project; post-project evaluation meetings; and commissioning specific 'learning reports'. However, stakeholder motivation is critical. Those involved will ask themselves, "Why should I tell you that?". Unless there is a good answer, they will keep their knowledge to themselves and not allow it to be captured.
3. **Transfer Knowledge**. Find ways to move the knowledge from where it is captured to where it is needed. Ideas include: have program staff train others; hold informal 'lessons learned' meetings that mix staff from the former program with others; create an email discussion list about e-government success and failure. Think of the transfer process as a 'knowledge market' with buyers and sellers of knowledge (the sellers being those who have learned from the previous programs). Motivation will again be critical. Sellers must want to sell their knowledge, and buyers must want to buy that knowledge. Are suitable incentives in place to encourage buying and selling?
4. **Apply Knowledge**. This is hard to organise. People will apply knowledge if it is useful to them; and won't if it is not. Build into new programs some fenced time that is allocated solely to 'learning lessons from the past'.

As much as we need to pursue all aspects of risk management, we need to most actively work on communication and changing the organizational culture to embrace knowledge form past programs.

Table 1. A Few Unsuccessful Programs Over The Years

Program	Date	Cause and Result
Molasses Tank Failure, Boston MA	1919	A 15-meter high tank burst unexpectedly, dropping two million gallons of molasses (in a 10-meter high wall, initially) into the streets, killing 21 and injuring 150. (reference 1)
Tacoma Narrows Bridge, WA	1940	Suspension bridge failed after resonating in torsion for some time under wind loading. The collapse brought engineers world-wide to the realization that aerodynamic phenomena in suspension bridges were not adequately understood in the profession nor had they been addressed in this design. (reference 2)
XSM-64 NAVAHO	1957	Program lacked a “test-as-you-fly” process that resulted in 10 flight test failures and 2 explosions. The result was a cost of $700M for a total flight time of less than an hour and cancellation of program (“Never go, Navaho”) (reference 3)
USS Thresher Submarine Sinking	1963	Silver-brazed pipe-joints passed hydro-proof, but failed to meet minimum bonding specifications when subjected to new ultrasonic testing technique. Navy failed to specify the extent of the new test requirement and did not confirm that the testing program was properly implemented. Ship passed on a “not-to-delay-vessel” basis. Pipe-Joint failure caused power failure and flooding in the engine room. Resulted in loss of ship and 129 lives. (reference 4)
West Gate Bridge, Melbourne Australia.	1970	Two sections forming halves of deck didn't fit together well. Holding bolts at end of bridge were loosened to allow for sections to be joined, and these bolts failed under environmental loads as sections were in process of being joined. One section thus collapsed, resulting in several fatalities. (reference 5)
Tomahawk/LASM/Naval Fires Control System (TLN)	1980	TLN requirements creep resulted in cost increases, schedule slippage and functional deficits. Failure to follow industry best practices resulted in poor software quality. Program cancelled. (Reference 6)
Hyatt Regency Skywalk, Kansas City MO	1981	Skywalk failed under live load, causing fatalities. Original design called for nuts where they could not actually be installed. (reference 7)
Union Carbide Piping Systems Failure, Bhopal India	1984	Thousands killed and injured as a result of toxic vapor leak. Several safety systems were out of service and plant was understaffed due to costs. (reference 8)
Space Shuttle Challenger	1986	Solid-fueled booster motor leaked combustion gases due to failure of a pressure

		seal (i.e., O-ring), leading to explosion of liquid fuel tanks and seven fatalities. O-ring had design problems that went unattended. Launch constraints were waived at the expense of flight safety. Safety margins were assumed based on previous successes. (reference 9)
THERAC-25 Cancer Irradiation Device	1987	Problems with software design led to radiation overdoses at some clinics. No safety prompts were designed, and device was essentially run open-loop. The failure itself had nothing to do with these points -- it was based on an obscure bug which required very fast data entry at very specific times. (reference 10)
Patriot Missile Radar System	1991	A software roundoff error increased monotonically after eight hours continuous operation. This was corrected, but not before a Scud missile killed several US Marines. (reference 11)
Hubble Space Telescope	1990	The curve to which the primary mirror was ground was incorrect, causing "spherical aberration". The flaw resulted in a fuzzy focus image. Result was four costly servicing missions in 1993-2002 and more than $3 billion. (reference 12)
Sleipner North Sea Oil Platform	1991	Platform supported by four elongated concrete cells, one of which sank during a standard blasting operation. There was an inaccurate finite element analysis performed on concrete support structure. The shear stresses underestimated by 47% and subsequently the concrete structure failed. The result was the loss of the platform at a cost of $1B. (reference 13)
Bank Accounting System	1996	Software bugs caused the bank accounts of 823 customers of a major U.S. bank to be credited with $924,844,208.32 each in May of 1996, according to newspaper reports. The American Bankers Association claimed it was the largest such error in banking history.
European Space Launcher – Ariane 5	1996	Launch failure caused by complete loss of guidance and attitude information 30 seconds after lift-off . Cause was specification and design errors in the software of the inertial reference system and subsequent reuse of Ariane 4 software and values. The result was a destroyed rocket and its cargo valued at $500 million. (reference 14)
Mars Polar Lander and Mars Climate Orbiter	1999	The root cause of the MCO failure was the use of data in English units that were thought to be in metric units within a segment of ground-based, navigation-related mission software. The loss of MPL has been traced to premature shutdown of the descent engines, resulting from a vulnerability of the software to transient signals. Failures were caused by a lack of end-to-end testing, "…inadequate checks and balances that permitted an incomplete systems test and allowed a significant software design flaw to go undetected." The results were the loss of both missions at a cost of $327M. (reference 15)
NOAA-N Prime Weather Satellite	2003	-Bolts securing the satellite to a turnover cart were removed -Satellite toppled from its platform during tilting operation. The causes were identified as a lack of discipline, not following procedures – Responsible Test Engineer (RTE), Production Quality Control (PQC), and Production Assurance (PA) were identified as the offices of primary responsibility. The result was a complete loss: $239 Million. (reference 16)
Electrical Power Outage, Northeast US	2003	A software bug was determined to be a major contributor to the 2003 Northeast blackout, the worst power system failure in North American history. The failure involved loss of electrical power to 50 million customers, forced shutdown of 100 power plants, and economic losses estimated at $6 billion. The bug was reportedly in one utility company's vendor-supplied power monitoring and management system, which was unable to correctly handle and report on an unusual confluence of initially localized events. The error was found and corrected after examining millions of lines of code. (reference 17)
Canadian Welfare Management System	2004	In July 2004 newspapers reported that a new government welfare management system in Canada costing several hundred million dollars was unable to handle a simple benefits rate increase after being put into live operation. Reportedly the original contract allowed for only 6 weeks of acceptance testing and the system was never tested for its ability to handle a rate increase.
Bank Accounting Software System	2004	Millions of bank accounts were impacted by errors due to installation of inadequately tested software code in the transaction processing system of a major North American bank, according to mid-2004 news reports. Articles about the incident stated that it took two weeks to fix all the resulting errors, that additional problems resulted when the incident drew a large number of e-mail phishing attacks against the bank's customers, and that the total cost of the incident could exceed $100 million.

References:

1. Catastrophic Tank Failures: Highlights of Past Failures along with Proactive Tanks Designs, John R. Cornell and Mark A. Baker, The US EPA Fourth Biennial Freshwater Spills Symposium (FSS 2002), Cleveland, Ohio, March 19-21, 2002.
2. http://www.me.utexas.edu/~me179/topics/lessons/case1.html
3. http://www.fas.org/nuke/guide/usa/icbm/sm-64.htm
4. A History Lesson: The Loss of the USS Thresher, Occupational Safety Observer June 1994
5. http://teachit.acreekps.vic.edu.au/cyberfair2002/Westgatebridge.htm
6. DOD IG Audit Report - Acquisition Of The Naval Fires Control System, Report No. D-2002-036 January 8, 2002
7. Kansas City Hyatt Regency Skywalk Collapse, by Alex Kieckhafer, Tony Moses, Andy Warta, December 7, 1999 (http://em-ntserver.unl.edu/Mechanics-Pages/Group1/INDEX.HTM)
8. http://www.bhopal.com/
9. SEASAT:Lessons Learned . . .And Not Learned, by Rick Obenschain, Acting Director of Flight Programs and Projects, NASA Goddard Space Flight Center, 5-20-2005
10. An Investigation of the Therac-25 Accidents, Nancy Leveson and Clark S. Turner, *IEEE Computer*, Vol. 26, No. 7, July 1993, pp. 18-41.
11. GAO/IMTEC-92-26, Patriot Missile Software Problem
12. http://www.uoguelph.ca/~ebaig/brian.html
13. http://www.ima.umn.edu/~arnold/disasters/sleipner.html
14. Lecture 22: Software Disasters, Kenneth M. Anderson Software Methods and Tools CSCI 3308 - Fall Semester, 2004
15. http://www.jpl.nasa.gov/marsreports/marsreports.html
16. NOAA N-Prime Mishap Investigation Final Report, NASA, September 13, 2004
17. Effects Of Catastrophic Events On Transportation System Management And Operations, Executive Summary Of The August 2003 Northeast Blackout Great Lakes And New York City, DOT-VNTSC-FHWA-04-05, May 2004