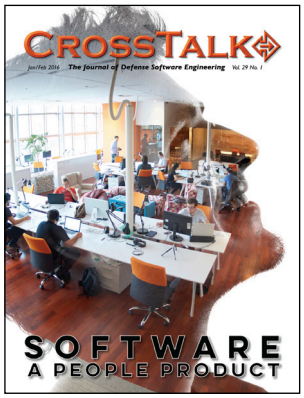
CROSSTALK

Jan/Feb 2016 **The Journal of Defense Software Engineering** Vol. 29 No. 1

SOFTWARE
A PEOPLE PRODUCT

TABLE OF CONTENTS

Departments



Cover Design by Kent Bingham

Software - A People Product

CrossTalk

NAVAIR Jeff Schwalb
DHS Joe Jarzombek
309 SMXG Karl Rogers

Publisher Justin T. Hill
Article Coordinator Heather Giacalone
Managing Director David Erickson
Technical Program Lead Thayne M. Hill
Managing Editor Brandon Ellis
Associate Editor Colin Kelly
Art Director Kevin Kiernan

Phone 801-777-9828
E-mail Crosstalk.Articles@hill.af.mil
CrossTalk Online www.crosstalkonline.org

CrossTalk, The Journal of Defense Software Engineering is co-sponsored by the U.S. Navy (USN); U.S. Air Force (USAF); and the U.S. Department of Homeland Security (DHS). USN co-sponsor: Naval Air Systems Command. USAF co-sponsor: Ogden-ALC 309 SMXG. DHS co-sponsor: Office of Cybersecurity and Communications in the National Protection and Programs Directorate.

The USAF Software Technology Support Center (STSC) is the publisher of **CrossTalk** providing both editorial oversight and technical review of the journal. **CrossTalk's** mission is to encourage the engineering development of software to improve the reliability, sustainability, and responsiveness of our warfighting capability.

Subscriptions: Visit <www.crosstalkonline.org/subscribe> to receive an e-mail notification when each new issue is published online or to subscribe to an RSS notification feed.

Article Submissions: We welcome articles of interest to the defense software community. Articles must be approved by the **CrossTalk** editorial board prior to publication. Please follow the Author Guidelines, available at <www.crosstalkonline.org/submission-guidelines>. **CrossTalk** does not pay for submissions. Published articles remain the property of the authors and may be submitted to other publications. Security agency releases, clearances, and public affairs office approvals are the sole responsibility of the authors and their organizations.

Reprints: Permission to reprint or post articles must be requested from the author or the copyright holder and coordinated with **CrossTalk.**

Trademarks and Endorsements: **CrossTalk** is an authorized publication for members of the DoD. Contents of **CrossTalk** are not necessarily the official views of, or endorsed by, the U.S. government, the DoD, the co-sponsors, or the STSC. All product names referenced in this issue are trademarks of their companies.

CrossTalk Online Services:
For questions or concerns about crosstalkonline.org web content or functionality contact the **CrossTalk** webmaster at 801-417-3000 or webmaster@luminpublishing.com.

Back Issues Available: Please phone or e-mail us to see if back issues are available free of charge.

CrossTalk is published six times a year by the U.S. Air Force STSC in concert with Lumin Publishing <luminpublishing.com>.
ISSN 2160-1577 (print); ISSN 2160-1593 (online)

CrossTalk would like to thank **309 SMXG** for sponsoring this issue.

We often focus on the tools required to develop software and it's true, there are many tools available that help develop requirements, design and test software. There are tools used to manage processes, configuration and even tools that help with software assurance and cyber security. But make no mistake, software is a product of people. In the end it's people that communicate their need, it's people that develop and refine the requirements, design the architecture, code the unit level programs, integrate the software modules, and test the software and processes for quality and functionality. There's no getting around it, it's a people product.

So, what are we doing to take care of the people? How do we prevent knowledge base from walking out the front door and signing on with the competition? How do we keep employees interested in our products, trained with the latest tools, languages, processes capabilities and management techniques to meet the challenges that allow for success? How do we continue to attract new talent to not only meet our existing demands, but keep up with our continued growth? One thing I have learned is if you are successfully developing software, then your business is growing and it's growing fast. Software is quickly taking over in terms of functionality. We rarely have to make hardware changes to add new capabilities; it's all done with software. Software is becoming more and more complex and the demand for MORE is the challenge facing all of us in the industry. These are the questions that keep me up at night while trying to successfully manage the Software Maintenance Group at Hill Air Force Base.

Not only do we have to figure out how to stay ahead of the game as we continue to move software from the art form, to an engineering discipline, we need to create a strategy for transferring the knowledge base from the grey beards to the young superstars and keep them interested enough to stay. Money is often considered the way to keep personnel from leaving, but more and more, our new rising superstars are more interested in the ability to work on products that interest them, work on technology that they find interesting, while working in an environment that's flexible enough to fit their lifestyle. The current market for software has created such a need that they can work almost anywhere they want. I recently saw a report indicating that there are four positions available for every software applicant we have. Those are TOUGH odds for the software industry.

The Software Maintenance Group at Hill Air Force Base is embroiled in a continual dilemma trying to overcome these obstacles. We constantly review policies to create an enticing environment for our employees. We recognize that getting "the right people on the bus" is crucial to our continued success. As a military organization, we certainly have our share of administrative constraints, just as all companies have their own challenges to overcome. We work hard to create the right mix of benefits, flexibility and positive environment for our people. Our goal is to create the "Best Place on the Planet to Work!"

The articles included in this edition of CrossTalk focus on Software – A People Product. I hope you enjoy them as you turn your organization's eye inward and focus on your people.

Richard L. Burnett
Deputy Director
309th Software Maintenance Group

Increasing Collaboration By The Minute

Alistair Cockburn, Humans and Technology, Inc.

Abstract. You can increase or decrease collaboration directly by specific actions. Once you learn to see these actions in practice, you can notice immediately when the rules are used, or broken, and watch how collaboration changes as a result.

Introduction

In 1971, Gerald Weinberg [1] described the central role of a soda vending machine at a university's programming help desk. The department administration, disturbed by the students congregating around the machine, ordered it removed. Much to their surprise, the line at the help desk suddenly became much longer. It took a while before they worked out that the students congregating around the soda machine were helping each other solve their programming problems, and thus reducing the load on the help desk staff.

Around 1974, while creating the first Visa credit card clearing system, Dee Hock [2] and his staff used an odd project management scheme: Working in a warehouse, they simply put on the large wall all the tasks they needed to get accomplished, according to the date it needed to get done. Someone hung a cup on a string to mark the current date. Each day, someone moved the string to the right, and everyone jumped on whatever tasks were now to the left of the string.

Forty years ago, these stories were mysteries. In 1995, Hutchins [3] describing how merchant marines bring their ships into port, used the term "distributed cognition" to describe how the crew operates as a though a single brain with distributed components.

This phrase, "distributed cognition," helps us understand why proximity and collaboration are so important on software projects [4]. Each person on the team is busy forming a slightly different idea of what problem they are solving, and how the solution should look. Each is running into problems that possibly someone else on the team might be able to help with.

Viewed in this way, we see software development as a problem of mental search and synchronization. We have to add the difficulty of learning how the other people work, what motivates them, what angers them, and so on, and the difficulty in resolving differing opinions as to whose view to accept.

We see, from this perspective, how it comes that communication and collaboration are so important on software projects.

The communication aspect has been heavily studied. Thomas J. Allen [4], studying (non-software) research and development teams, found that communication drops off at about 10 meters in distance (basically, people won't walk longer than the length of a school bus to ask a question). Olson and Olson [5] describe the nature of the productivity gain of collocated teams over distributed teams.

However, the matter of collaboration is not so clear cut. A search for "increasing collaboration" turns up more than 200 million results. The articles at the top of the list come from Forbes [6], Harvard Business Review [7], and similar. Here are the suggestions from the first two:

- Start a tradition
- Create a Board of Awesome
- Walk
- Eat right
- Don't be late
- Smile
- Take regular breaks
- Breathe
- Nap
- Get executive support
- Invest in signature relationship practices
- Model collaborative behavior
- Create a gift culture
- Provide training for collaboration
- Create informal communities
- Assign leaders who are both task- and relationship-oriented
- Build on heritage relationships
- Clarify roles and tasks

All of those are no doubt good and useful. However, I often find myself in a meeting or collaborative session and wondering,

"What can I specifically do, now, to make this session go better (with respect to gathering everyone's insights and contributions)?"

As the meeting rolls along, the level of collaboration and contribution may change for the better or the worse. I wonder, "What triggered that?"

The question I wish to address is, what specific actions can people take to increase collaboration on a minute-by-minute basis. What induces people to collaborate more?

Enabling Bravery

In 2007, I conducted a small grounded-research study to address that question [8]. The raw results are posted online [9], so that others might reach different conclusions from the data I gathered.

In what follows, I highlight aspects of that study, what I have learned since, and how the reader might add to the list.

Based on the study, when I watch a group collaborate, I see the following:

- One person assumes enough bravery to claim the stage.
- Everyone else yields to that person.
- The speaker offers personal insights to the others.
- The speaker relinquishes the stage, opening it for someone else.

In that short sequence, the first is the most amazing. In deciding to speak, the person has to conclude:

"What I have to say is more important than what anyone else has to say, and they need to all be quiet and listen to me."

For many people, that is a frightening proposition. It is a claim of ego, and fraught with potential embarrassment.

As though watching a movie, I see a friendly game of Whack-A-Mole [10] (without the hammer, of course). Different people take turns standing up, talking, sitting down. In a good collaborative session, everyone takes a turn standing. In a poor collaborative session, only one or a few contributors stand, the others stay seated, their insights remaining lost to the group.

The breakthrough in my understanding of the raw data in my study came from the book "Impro," by Keith Johnstone, an acting trainer [11]. He describes how we immediately understand being above or below someone in a social hierarchy, and how our body and behavior changes as a result.

Seen this way, the Whack-A-Mole image is remarkably appropriate. Each person has to assert social superiority for a moment in order to contribute. How can we get all the timid people to do this, and how can we get all the dominant people to leave enough space for them to do so?

This turns out to be the central aspect of collaboration in this one study.

Specific Actions

With the help from some friends and colleagues, I was able to mine the data to extract several dozen specific actions that seemed to change the immediate state of collaboration. I put them into four categories:

- Lift Others
- Increase Safety
- Get Results
- Add Energy

It is important to note that the Whack-A-Mole image only captures the first two categories. But then a woman reader wrote:

"When I have a sympathy session with my girlfriend, we lift each other all the time, and we have all the safety we need. Are we collaborating?"

From that question came the need for the third category, Get Results. Without results, the session might have been agreeable, but is not what we would consider as "collaboration."

The fourth category came from looking for additional actions still not covered by the first three. It is possible there are more major categories, these are the ones I have to this point. Further in this article, I describe how to add your own recommendations to the list.

The list has proved very effective in decoding collaboration sessions. As we got used to noticing movements people made according to the list, we could see instantaneous changes in the group's mood. As participants, we could help defuse a negative action someone might have made with a counter-action to help restore a collaborative mood. I, personally, became very sensitive to when I unwittingly did the opposite of what the list said to do. I could see one or more other people shrink down and decide not to contribute for a bit. In short, the list turns out to be accurate, useful, and actionable, both in the positive and negative versions.

Collaboration Cards

Having the list on paper or in an article was sufficient for me, but did not spread well to other people. So I created a deck of "Collaboration Cards" [12] for others to learn from. While still not perfect, the cards allow people to study one or two actions at a time until they learn to recognize their being enacted or violated, by themselves or other people.

Here is list of actions in the current set of Collaboration Cards, with some additional notes on specific ones.

Note that these actions are not just for the session leader or facilitator. They can be used by every person in the session.

As a reader, you might look for which one is your preferred mode of operation in a collaborative session, and which one is most difficult for to you enact.

Lift Others

This is possibly the most important category, since what we are trying to do is get people to step forward when they might be timid.

• Lower Your Relative Social Position

By tone of voice and gesture, place the other person at your same level or higher. This includes self-deprecating humor. It does not mean groveling.

Commentary: This is the keystone action coming from the book, "Impro." Watch as someone bows their head when they speak, or literally shrink their body, to indicate their temporary reduction in status. This is most effectively used by people in important social positions.

• Recognize Others

Ask for their thoughts, accept an idea. When you build on their idea, let them know, so they get recognition. Delight in the ways they find to implement their ideas.

• Inquire, Don't Contradict

When inclined to contradict, inquire instead, to discover new information that makes the answer other than what you expected. Work to understand why the other person's answer is so different.

• Challenge but Adopt

It is uplifting when someone disagrees with you at first but then sees and adopts your view. Do this for someone else. Look to adopt their ideas where possible, so they know they are heard and their ideas valued.

Increase Safety

If "Lift Others" lets people operate from where they are, "Increase Safety" expands the collaborative area. As such, it is potentially more dangerous when you get it wrong.

• Be Yourself

People can usually tell if you are being yourself or acting. Being yourself shows there is nothing to be afraid of. Try "being in the bar at 9 p.m. with friends," quite obviously relaxed and your regular self. (This is not an excuse to be crude.)

Commentary: "Being in the bar at 9 p.m. with friends" is a potentially dangerous move. My colleague Jeff Patton phrases it this way: "There's this person in a suit with his Blackberry, messaging away, and suddenly he notices he's not in a meeting room any more, but in the bar at 9 p.m. with friends, and he puts his Blackberry away and joins the discussion."

• Say Something Honest, On the Edge of What You Think is Allowed

Say or do something that you would like to, but which might lie outside the expected boundaries. This widens the boundaries of what others can do. What others were afraid to say or do may suddenly appear "safe" to them.

Commentary: This is the most dangerous move in the list. Unfortunately, it happens to be my specialty. Jeff Patton, comments again, "By the time you get done violating all social decorum, and everyone is having a great time, suddenly those little obstacles other people were having look tiny in comparison, and they start to contribute." And of course, when I get it wrong, it is embarrassing.

• Add Humor

Humor lowers tension, allows relaxation. It is not the making of a joke that increases safety, it is that safe groups feel safe joking with each other. Personal attacks disguised as jokes do not count.

• Show You Won't Hurt

Show that you won't say things that hurt the other person. With someone to back up and protect them, a person might feel brave enough to step in and contribute.

• Leave Some Privacy

If there is nowhere safe to hide, fear goes up and safety goes down.

• Don't Leak Information That Will Hurt Someone

This should be obvious.

Get Results

There are different forms of "result" that improve the session.

• Get One Result

Getting a result is heartening. Good facilitators often generate a victory to help encourage and bind the group. If the session is ending, aim for a small goal, so that the group can end with a victory.

Commentary: A collaboration session is not a collaboration session without results. Getting a result, all by itself, changes and improves the texture of the collaboration. Some astute leaders and facilitators will specifically search for and arrange for the group to share a "win" either early in the day, or to save the group from depression at the end of a long, fruitless day.

• Say Something Valuable

Try to make your first speaking of value. This moves the work forward, and it encourages others to listen to you.

• Get Back From Diversions

Keep your ideas on topic. Going off track for a little while releases some tension in the room, but people appreciate being brought back.

• Clarify the Way Forward

Sometimes it helps to "pull the threads together," show what has been achieved, what forward looks like, or where the group is.

Add Energy

The final category addresses such things as posture while listening, or ways of inject new energy.

• Keep Your Energy High

Avoid being lethargic yourself. Body posture, muscle tone, eye alertness, all communicate your energy level. Even just sitting alert contributes energy to the room. Pay close attention to the speaker, digest what they say, ask a question.

• Contribute

Contributing your own ideas adds energy to the room. If everyone only sits and listens, the group will wind down. When people see that you are not afraid to give away your ideas, they also feel safer in offering up their own.

• Challenge

Challenge others' ideas. Not to put people down, but to explore the truth and limits of the ideas. Challenging an idea is part of being honest, listening intently, and making progress.

Commentary: This is the other potentially dangerous action in the list, and needs to be used with some care. There are people who challenge all the time, and become viewed as a nuisance to the group. On the other side, I have come to notice the following scenario: The group is tired or bored, the speaker is droning on. People are slouched in their chairs, waiting for the speaker to be done and the pain to be open. Suddenly, one of them hears something interesting, leans forward, and asks a question about or challenges what the speaker just said. In a moment, everyone wakes up, sits forward, and listens. At this moment, collaboration has started again.

Using the List

We quickly learned that it is too difficult to hand the list of actions to everyone and ask them to notice the behavior of the group while also participating in the session. Whether it was with a list or cards made no difference.

What worked was to give each person just one card or item from the list, and ask them only to notice occasionally when it was being used or violated. Variations on this idea include asking them to make tick marks on a paper when they see it used, other tick marks for violated. The important thing is not to take too much of the person's attention away from the content of the meeting.

One pair of trainers who train upcoming facilitators hand out one card in a facilitation session led by other students, and ask them to watch their one card in action or violation during the session. They then trade insights afterwards.

One town manager gave the cards to her division supervisors, the police chief, the fire chief, chief of sanitation, and so on, for them to use with their subordinates. She was less concerned with collaboration inside a single meeting, than with building a culture of collaboration over the long term. Her insight was that the same actions have longer-term effects as well as in the moment.

Some people adopt a habit of carrying around one card with them each day, so they can become sensitive to that one item in many settings, without having to focus on it all the time.

One person put a different card on his car windshield visor day, as a form of passive learning as he drove to work.

One used that has been proposed, but not yet applied, to my knowledge, is to video a meeting or collaboration, without any use of the list of actions, and then to review the video afterwards, using the list. In the review of the video, everyone would have the entire list at hand, and would call out when an action enacted or violated an item on the list. They could then replay and examine that moment on the video, and decide what they, as a group, wanted to learn from the moment.

This video-and-replay technique would be a good way to notice additional actions, not on the list, that also contribute to improved collaboration.

Discover More Yourself

The list is obviously not complete. I believe it would be a good exercise for a group to personalize it by creating their own addenda to the list.

Here is the technique I used to create the list in the first place, adapted to a group adding to it:

- Have a meeting or collaborative session as normal, but ask people to notice at what moments the mood to collaborate increased or decreased.
- Write down in detail and objectively what happened just before and just after that moment.
- Now comes the hard part: attempt to decode what caused the shift in mood. What underlying action made the difference at that moment?
- Give is a cute, short, verb name. Use the imperative voice, so it is a "Do This" type of a phrase.
- Watch it in action, and see if it actually makes a difference, and if violating it causes a loss in collaborative mood.
- When you have a good addendum, publish it online for others to experiment with.

Summary

It is possible for individual people in a collaborative session to directly affect the mood for collaboration for better or for worse, minute by minute.

This article listed 17 specific actions, in four categories:

- Lift Others
- Increase Safety
- Get Results
- Add Energy

Enacting those actions tends to increase the mood for collaboration, violating them tends to decrease it.

The list is, of course, not complete. Each group might profit from adding to the list as its own form of learning and personalizing the actions that improve collaboration.

ABOUT THE AUTHOR

Dr. Alistair Cockburn, one of the creators of the Manifesto for Agile Software Development, was voted one of the "The All-Time Top 150 i-Technology Heroes" in 2007 for his pioneering work in use cases and agile software development. A renowned IT strategist and author of the Jolt award-winning books "Agile Software Development" and "Writing Effective Use Cases," he is an expert on agile development, use cases, process design, project management, and object-oriented design. In 2001 he co-authored the Agile Manifesto, in 2003 he created the Agile Development Conference, in 2005 he co-founded the Agile Project Leadership Network, in 2010 he co-founded the International Consortium for Agile. Many of his articles, talks, poems and blog are online at <http://alistair.cockburn.us>.

Email: totheralistair@aol.com

REFERENCES

1. Weinberg, G., The Psychology of Computer Programming: Silver Anniversary Edition, Dorset House, 1998.
2. Hock, D., Birth of the Chaordic Age, Berrett-Koehler Publishers, 2000.
3. Hutchins, E., Cognition in the Wild, MIT Press, 1995.
4. Allen, T., Managing the Flow of Technology: Technology Transfer and the Dissemination of Technological Information Within the R&D Organization, The MIT Press, 1984. <http://www.ics.uci.edu/~corps/phaseii/OlsonOlson-DistanceMatters-HCIJ.pdf>
5. Olson, G., Olson, J., "Distance Matters", HUMAN-COMPUTER INTERACTION, 2000, Volume 15, pp. 139–178.
6. <http://www.forbes.com/sites/ekaterinawalter/2013/12/23/how-to-reduce-stress-and-increase-internal-collaboration-in-the-workplace/>
7. <https://hbr.org/2007/11/eight-ways-to-build-collaborative-teams>
8. <http://alistair.cockburn.us/Collaboration%3a+the+dance+of+contribution>
9. <http://alistair.cockburn.us/Collaboration%2c+the+Dance+of+Contribution%2c+raw+notes>
10. <https://www.youtube.com/watch?v=GVJL9oXgsAA>
11. Johnstone, K., Impro: Improvisation and the Theatre, Routledge, 1987.
12. <http://alistair.cockburn.us/Collaboration+Cards>

The Capability Conundrum: Creating Constructs for Unleashing the Power and Potential of Your Most Important Resource

Jonathan Powell, UMBC and CACI

Abstract. In the foreseeable future, humans remain key to software systems development. Therefore, emphasis should be placed on how we maximize both the potential and productivity of what is in fact the most critical element of the Software Development Life Cycle (SDLC).

Introduction

Four years ago in this Journal I wrote:

"Today when the Navy needs next generation software for its submarine sonar systems, defense contractors are not deploying hordes of automatons to the Pentagon to gather requirements, design the software, build the prototype, make it ready for production, and then support it through operations and maintenance. It still comes down to people, and I submit it will come down to people for a long time to come. Even in the far out future, as parts of this chain are automated, people will be needed to intercede, because software is not perfect, and problems always arise. So if one's ability to overcome software challenges fundamen–tally comes down to people, the question becomes, 'How do I get the most out of my people?' This will be the focus of this article."

Indeed, not much has changed on that front. There may indeed come a time when humans are removed from the Software Development Life Cycle (SDLC) in significant measure. But in the foreseeable future, humans remain key to software systems development. Therefore, emphasis should be placed on how we maximize both the potential and productivity of what is in fact the most critical element of the SDLC. With this in mind, this article will examine how do we enhance the quality of software talent. And how do we improve innovation, training, and motivation when it comes to human capital.

Attracting Talent

At a macro level, in the Aerospace and Defense Industry (A&D), we need to improve our ability to attract talented individuals who pursue careers in computer science, cybersecurity, and related fields. This is particularly challenging in an era where Google and Facebook are booming, and offer to college graduates the promise of riches and the opportunity to work on cutting edge technology and innovation. While the landscape is challenging, is doesn't have to be daunting. In certain sectors of A&D (parts of Defense, for example) there may be opportunities to expose prospective candidates to new technologies. However, what A&D really needs to do in order to make substantial gains in attracting and retaining top talent is "hook" them early. This is done by getting them involved doing mission work. For many people, the value of meaningful work outweighs money, new technologies, and other factors. And it is hard to get more meaningful than supporting our Nation's work catching bad guys and terrorists. There are a whole host of other meaningful areas in A&D as well, not just limited to National Security or Defense. Exposing our young talent to these areas and "setting the hook," imbuing them with the thrill of pursuing something higher than themselves, is what we're after. The characteristics of this work are where we have an advantage over many commercial players. The younger the age where the hook is set, the greater the chance of inspiring an enduring, even lifelong motivation to serve in mission oriented roles. This can be done through College Internships, CO-OPs, and the like. But younger is better, even offering internships at the high school level or education and exposure opportunities at ages younger than high school are important. This is especially true since the Facebooks of the world have an inherent advantage with their name brand and clout, and often the important and interesting mission work we perform is unknown to the general populace. Today, we in A&D have the opportunity to go for the "two-fer" – not only entice the younger generation with the attraction of "cool" mission-oriented work, but link this work to the "cool" technologies in the market or coming into the market, positively impacting the mission. A perfect example of this is 3-D Printing, and the revolution in design it is sparking. Take a kid who's good with computers, knows about 3-D Printing because 3-D Printing is used to build cool custom Lego parts, and introduce him or her to the cool applications that are being used in the Defense community, and bingo – game, set, match. And if that kid doesn't "bite," others will, and at a greater rate than through the traditional model of competing with Silicon Valley and others attempting to hire these students after they've graduated college. One other point for consideration here – the world has changed so now young talent can go work for a "cool" company and still have a mission focus (look at what Amazon is doing for the Intelligence Community). I would expect that the increase in commercial players in mission oriented work will continue – DoD has even made this a point of focus by standing up a Silicon Valley office. This is a positive development, and will only help get the word out on the important work and opportunities that exist in the A&D sector. This means traditional A&D players will need to work harder and come up with different innovations to continue to spark mission interest in youth in order to successfully compete for talent against non-traditional players in the space.

Innovation and Motivation

So the conversation still begins and ends with people. Not as pithy statements or some platitude, but a compass to follow. If the organization makes hiring and retaining the right talent its true guiding light, then innovation can surely follow. While having

knowledgeable and skilled resources in place is a foundation, by itself it is not enough. The organizational culture must encourage and enable innovation. Policies and procedures can support innovation – for example, giving people free time at work "off the clock" to innovate, experiment, and think. These policies and procedures are especially needed in professional services organizations where there's usually not a formal R&D organization, in which staff is formally allocated significant time to pursue special projects and experiments. Financial incentives can help entice employees to come up with new ideas and bonuses and the like for ideas that are reviewed and accepted could be codified in policy. However, perhaps the single most important determinant in promoting an organization where innovation thrives is through setting the conditions for open and honest communications. This has to start with the leader of the Organization – His/her behavior will permeate all aspects of how business is done, and either positively or negatively impact the "Corporate Culture," defined as "The way people behave from moment to moment without being told" [1]. An Open and Honest Culture, often referred to as "transparency," has a multiplier effect, positively impacting not just innovation, but also morale. Joyce Russell, Vice Dean at the R.H. Smith School of Business writes, "When a firm does not have trust or transparency, not only are there employee problems (low morale, productivity, commitment and loyalty), but also employees will undoubtedly pass that along to their clients or customers by treating them poorly. On the other hand, firms with a transparent culture are more successful since employees feel free to come up with more creative solutions, they share issues before they become major problems, and they are more engaged, motivated, and productive at work" [2].

Note earlier I wrote knowledgeable and skilled resources, not necessarily what's coined as "experience" (i.e. number of years of service). This is a bias, particularly in A&D, where experience can be viewed as a prerequisite or barrier to entry, instead of just one piece of the individual's mosaic. For certain functions, there may be a benefit in having "experience" act as a gate-keeper prior to contribution, but innovation is not one of them. Case in point – some senior engineers in our intelligence sector wrestled with a thorny issue for months, without finding a solution. Around that time, in came a couple of Virginia Tech Co-Ops and they literally had the problem solved in a couple of weeks. Why? Were they smarter? Better educated? No and No. The answer simply came down to the fact these two individuals had a completely different way of looking at things and brought that lens to the problem, yielding a rapid breakthrough. This is a single vignette, but its implications are telling. Silicon Valley recognizes this fact, and notoriously does not give the "gray hair" any more credence than the new kid on the block. It is the best idea that matters and the one Silicon Valley and the market rewards. More of this attitude needs to be injected into A&D. In fact, this sort of cross pollination of talented millennials and younger generations with the established "gray hairs" is needed if A&D is to stay viable.

Training

For Cyber Security training, there are two facets. First, recognition of the fact there are plenty of well-trained folks out there. But often times, these folks can't be hired in A&D because of barriers (for example, inability to get a security clearance because of some disqualifying item from the past) or they have no interest in working within the A&D arena – the white hat or ethical hacker may be a category of individual who's not necessary inclined to work in the A&D space because they may perceive entities in this space as being stuffy, bureaucratic, etc. Organizations like the FBI are making changes in their clearance processes, to widen the aperture of qualified candidates who can apply and be accepted. In addition to looking at policies and procedures like the FBI is doing, organizations can attempt to be innovative. For example, an A&D company may elect to form an independent Skunk Works

subsidiary, possessing a Google-like entrepreneurial bent and freedom, that talented individuals like the ethical hacker may be enticed to join. This can be challenging, because the organization has to derive a way to get the technical gems out of the skunk works and into the MILSPEC world of the end user. But where there are risks, there are rewards, and the potential rewards for successfully implementing such model could be huge for the A&D firm. Second is the case where training is in place and needs to be improved. Of course, many of today's experts in Cyber Security are self-taught or learn through non-traditional means. There ought to be a way to recognize the talents these people bring to the table. So today where a degree or certification is required, instead look at experiential or other substitutes. The Army is establishing a Cyber Branch (an entire field, akin to Infantry, Artillery, etc.). They should give serious consideration to non-traditional means to fill this branch, especially since they need to get going and reach critical mass of personnel immediately. So the tried and true ASVAB exam, and picking those who score highest for some form of follow-on training, won't get it done. However, what could work is offering bonuses to resources with a proven skillset (for example, network penetration resources, hackers, etc.) without barriers (i.e. not making a degree, certification, etc. a requirement), and have the entry procedure involve passing a one-on-one interview with a slate of known cyber experts to see if the resource really does have the skillset they portray. There's a lot of talent literally sitting in the basement of homes across America (ala the classic movie 'War Games') just waiting to be tapped, and the old pipeline approach simply won't get it done. In situations where training is needed and is provided, one enhancement has to be increasing the hands on, in-person elements. Cybersecurity is a space that requires on-the-job-training (OJT), where one can learn side-by-side with an experienced mentor or have the opportunity to experiment in a lab environment. I've heard numerous complaints about college or other offerings possessing high lecture content and low hands-on aspects, thus resulting in insufficient learning. Generally, anything that can be done to increase the hands-on aspects of training will improve the quality of the training and the curriculum. Note, here I am talking about the Doers – the Network defense, Pen Testers, Ethical Hackers, etc. Classroom and lecture are fine for policy makers, management, etc. But A&D is in dire need of Doers, and the way to increase the real pool of doers (different than the perceived pool of doers – those with certifications and degrees but aren't capable of performing hands on Cyber Security functions) is to ramp up the hands on aspects of training. There's not a need to be prescriptive in this regard. Just as long as the hands get on the keyboard in engaged and meaningful work, under the tutelage of someone who knows what they're doing, the learning will occur and most importantly, stick. And once this new pipeline of effectively trained folks is in place, one benefit is the assurance more of these folks will be effective in their roles sooner, versus the current scenario where people with degrees and certs are hired, but a significant portion are subsequently shown the door because they weren't the Doer they were perceived to be.

Conclusion

Software remains a product which is reliant upon human capital. Action needs to follow in order to nurture this critical resource, especially in sectors like A&D, if it is to remain viable in the future. While some companies and areas of government are making steps in adjusting the frameworks for attracting, retaining, training, and motivating software talent to meet current needs and those of the future, not enough is being done. As a first principle, let's take a look at our existing organizations, and ask ourselves how do we establish the conditions to harness and unleash the true array of human capital already available and bring these capabilities to bear. MIT Professor Alex Pentland sums this up nicely.

"It is not simply the brightest who have the best ideas; it is those who are best at harvesting them from others. It is not only the most determined who drive change; it is those who most fully engage with like-minded people. And it is not wealth or prestige that best motivates people; it is respect and help from peers" [3].

ABOUT THE AUTHOR

Jonathan W. Powell, CGFM, PMP, Security+, is an Adjunct Professor at UMBC and a Senior PM at CACI. A for¬mer submarine officer, Mr. Powell has led complex engagements for military, federal, and intelligence agencies. His articles have been published in PM Network and Con¬tract Management, respectively. Mr. Powell serves on the Board of the Montgomery County Revenue Authority, where he is a member of its finance committee. He holds a B.S. from USNA and an MBA from the University of Maryland.

E-mail: jopowell@caci.com

REFERENCES

1. See <http://fortune.com/2015/03/05/perfect-workplace/>
2. See <http://www.washingtonpost.com/news/capital-business/wp/2015/04/02/career-coach-the-importance-of-being-more-transparent-at-the-office/>
3. See <http://fortune.com/2015/03/05/perfect-workplace/>

The Tragedy of the Commons: Establishing a Strategic Project Management Office (PMO)

Peter D. Morris, PMP, PMI-ACP

Abstract. The Tragedy of the Commons is a dilemma arising from a situation in which multiple individuals, acting independently and in their own self-interest, will ultimately deplete a shared limited resource, even when it is clear that it is not in the group's long-term interest for this to happen [1].

Introduction

"100% of everyone's time should be taken up by projects."
-Tom Peters, Circle of Innovation

Popular theories about the Tragedy of the Commons hold that government or organizational intervention and private property are the only effective methods to prevent finite resources from being ruined or depleted. This parable was popularized by wildlife biologist Garrett Hardin in the late 1960s, and was embraced as a principle by the emerging environmental movement.

Indiana University professor Elinor Ostrom won the Nobel Prize for Economics for her theory that cooperative behavior greatly boosts the legitimacy of the commons as a framework for solving our social, environmental and personal advancement problems. Dr. Ostrom died in June, 2012 with the distinction of being the first and only woman to win the prize, although never formally trained in economics (she was technically a political scientist). Over many decades, Ostrom has documented how various communities manage common resources—grazing lands, forests, irrigation waters, fisheries—equitably and sustainably over the long term. "When local users of a forest have a long-term perspective, they are more likely to monitor each other's use of the land, developing rules for behavior," she cites as an example. "It is an area that standard market theory does not touch [2]."

Hardin himself later revised his own view, noting that what he described was actually the Tragedy of the Unmanaged Commons. Ostrom's research refutes the Tragedy concept with the real life experience from places like Nepal, Kenya and Guatemala. A classic example of this is an acequia, a centuries-old tradition of a cooperative irrigation in New Mexico and Colorado where the small flow of water available for agriculture is allocated by the community as a whole through a democratic process.

The Tragedy of the Commons might be equally applied to common project management practices where individual projects are siloed, acting almost as separate entities in organizations that would benefit from cooperative behavior, long-term perspectives and the development of rules for behavior. Limited project resources such as people, funding, equipment, facilities, tools, network capacity and expertise replace lands, forests and water, but are no less important to the success of a project portfolio.

As a newly minted project manager with the DISA Joint Spectrum Center (JSC) back in the late eighties, I sought out the one PM most admired for her estimation skills. I asked Elissa what was her process? She said she simply asked her developer how long it would take to develop the software, then tripled that number, explaining that developers tend to estimate only their own time (which was about one-third of the total lifecycle time) to the exclusion of planning, requirements, testing, release and documentation activities. While comprehensive studies later confirmed that the development stage typically takes up 35% of the total [3], it would have been nice to have a common repository maintained by a PMO to discover this best practice. Much like the Commons, where challenges are limited to physical resources, organizations need a central entity to support project management. It would have also been helpful at the JSC to have a PMO where training, coaching and information were available in configuration management, scheduling, metrics, monitoring, reporting and management of risk, issues and decisions. The challenge for any successful PMO, as with the Commons, is to provide these resources and to prove without doubt that the project manager will benefit from similar PMO services.

Most organizations have established a number of groups, programs, initiatives, meetings and activities targeting project management control and excellence. These may include structures dedicated to learning and applying project management principles, and various status meetings and retrospectives where progress, quality, budgets, metrics, risks, issues and other important subjects are discussed and documented. The primary goal is to promote solid project management processes, methodology and culture. As companies begin to recognize the favorable effects that project management has on profitability, emphasis is now being placed on achieving professionalism in project management using the project office concept. Even with new process improvement practices in place, maturity and excellence in project management does not occur simply by using project management over a prolonged period of time. Rather, it comes through strategic planning for both project management and the project office [4]. Or to paraphrase Dr. Ostrom, when projects have access to limited organizational assets, project-level strategic planning with a long-term perspective will allow for monitoring of overall objectives, common tools and services, project metric and other repositories, dashboards, facilitators to support detailed and predictable planning with desired results, new and more efficient estimation techniques, cost effective organizational training and the development of rules for behavior. Successful PMOs realize that it's not enough just to deliver value—they make the business value of the PMO known throughout the organization, consistently and often [5].

Background

While organizations currently monitor or measure numerous project activities, the only clear way to have a global sense of how projects are doing is to have a defined project focus point: the Project Management Office (PMO). A Gartner Group study [6] predicted that companies that fail to establish a formal PMO will experience twice as many project delays, overruns and cancellations as will companies with a PMO in place. A more recent Gartner study states "It brings structure and support to evaluating, justifying, defining, planning, tracking and executing IT modernization efforts. It also encourages more business-side participation in IT modernization efforts and in the resolution of conflicts caused by limited resources and other constraints [7]." If Steven Covey is right, and interdependence is the name of the game, then the PMO is the way for people throughout the organization to recognize and capitalize on their interdependencies, to best manage and transfer project management knowledge, and to get into step with each other for the benefit of all [8].

Once repeatable and managed processes are established through Business Process Initiatives (BPI) and continuous process improvement programs, initiation of a PMO would seem to be the logical next step to ensure these programs are built to last and that positive organizational innovations continue. Using the theories of Jim Collins and other researchers, this article will present a justification for the establishment of a strategic PMO, along with a systematic approach to create and sustain project management value through an institutionalized PMO function.

PMO Concepts

"The way a team plays as a whole determines its success. You may have the greatest bunch of individual stars in the world, but if they don't play together, the club won't be worth a dime." -Babe Ruth

Although companies have employed PMOs since the mid- to late 1990s, the vast majority of PMOs have either been recently created or restructured [4]. A PMO's effectiveness and success depends on choosing which functions to implement, and adapting them and adjusting them to fit the organization's needs [9]. In a recent 3-phase program, it was determined that the relative differences in importance of various individual functions "reinforce the need to adapt to the organizational and strategic context when deciding which functions to include within the mandate of a particular PMO [10]." This research provides the most extensive list of the functions PMOs perform in organizations today. The 500 respondents in the survey were responsible for a variety of roles, but most were project managers or PMO members. Respondents ranked the importance of each PMO function on a scale of 1 to 5 (not important to very important), and the study ultimately identified 27 functions that PMOs can perform. Factor analysis grouped these into six distinct groups as indicated in Table 1. Table 2 lists these same PMO functions in decreasing order of industry importance, according to the same survey. While this order of importance will likely change for a given organization (e.g., a lessons learned or risk database might be closer to the top than the bottom of the list), it is instructive to note how industry as a whole prioritizes these functions.

Monitoring & Controlling Project Performance	**Development of Project Management Competencies & Methodologies**	**Multiproject Management**
• Report project status to upper management • Monitor & control of project performance • Implement & operate a project information system • Develop & maintain a project scoreboard	• Develop & implement a standard methodology • Promote project management • Develop competency of personnel, including training • Provide mentoring for project managers • Provide a set of tools specific to project needs	• Coordinate between projects • Identify, select and prioritize new projects • Manage one or more portfolios • Manage one or more programs • Allocate resources between projects
Strategic Management	**Organizational Learning**	**Other Functions**
• Provide advice to upper management • Participate in strategic planning • Benefits management • Network & environmental scanning	• Monitor & control PMO performance • Manage project documentation archives • Conduct post project reviews • Conduct project audits • Implement & manage lessons learned database • Implement & manage risk database	• Execute specialized tasks for project managers • Manage customer interfaces • Recruit, select, evaluate and determine salaries for project managers

Table 1. Importance of 27 PMO functions grouped into factors; n= 500, Hobbs & Aubry [2007]

In Good to Great [11] Jim Collins suggested that sustaining value generated from organizational investments requires both "preserving the core" and "stimulating progress." "Preserving the core" explains why appropriate on-boarding of experienced project managers is so important. His philosophy of first defining "who" before determining "what" speaks directly to the importance of preserving the core ideology as an anchor point while stimulating change, improvement, innovation and renewal. When we maintain a steady culture of discipline, we can give our employees more freedom to experiment and find their own best path to results. It also explains why PMOs should not get distracted from their primary focus (their hedgehog principle—see what is essential and ignore the rest), by taking on other responsibilities not part of their primary charter.

The PMO must stay focused on managing projects in order to continue to add value to the company. This in no way infers stagnation or neglecting core principles. As Collins observed, "If you successfully apply these ideas, but then stop doing them, you will slide backward…the only way to remain great is to keep applying the fundamental principles that made you great."

PMO Function	% of PMOs Where Important
Report project status to upper management	83%
Develop & implement a standard methodology	76%
Monitor & control of project performance	65%
Develop competency of personnel, including training	65%
Implement & operate a project information system	60%
Provide advice to upper management	60%
Coordinate between projects	59%
Develop & maintain a project scoreboard	59%
Promote project management within organizations	55%
Monitor & control PMO performance	50%
Participate in strategic planning	49%
Provide mentoring for project managers	49%
Manage one or more portfolios	49%
Identify, select and prioritize new projects	48%
Manage project documentation archives	48%
Manage one or more programs	48%
Conduct project audits	45%
Manage customer interfaces	45%
Provide a set of tools specific to project needs	42%
Execute specialized tasks for project managers	42%
Allocate resources between projects	40%
Conduct post project reviews	38%
Implement & manage lessons learned database	34%
Implement & manage risk database	29%
Benefits management	28%
Network & environmental scanning	25%
Recruit, select, evaluate and determine salaries for project managers	22%

Table 2. Identified PMO functions in decreasing order of industry importance, Hobbs & Aubry 2007

Collins' "Stimulate progress" theorem has application to several PMO-related issues, including developing project managers by having them take on more challenging roles or different types of projects, periodically updating or refreshing the project management methodology, or adding new functions to the PMO that enhance its ability to manage projects on the organization's behalf.

The Project Office: 1990s and 2000s

Project management competency represents important intellectual property, and therefore must be managed wisely. This responsibility is most appropriately that of the PMO [4]. After reviewing the roles and responsibilities of the project office over several decades, Kerzner listed PMO benefits in the 1990s and 2000s. As indicated in Table 3, the purpose of a PMO has changed with the business environment over the past two decades.

With the dawn of the 21st century, the PMO became commonplace in the corporate hierarchy. While the majority of PMO activities had not substantially changed, there was now a new mission for the PMO:

- Responsibility for maintaining all intellectual property related to project management
- Active support for corporate strategic planning

In the last decade the PMO began servicing the corporation, especially the strategic-planning activities, rather than focusing on a specific customer. The PMO was transformed into a corporate center for control of project management intellectual property. This was a necessity as the magnitude of project management information grew almost exponentially throughout organizations.

Note that the newer benefits of Table 3 relate specifically to project management processes and procedures. It is absolutely essential for an organization to establish mechanisms (intranet, project team sites and databases, other information systems) to capture this data and then disseminate the data to various stakeholders. Since much of this information is important for both project management and strategic planning, strategic planning for the PMO is imperative.

The Project Office: 2008-Present

Since the seminal work of Hobbs & Aubry no survey has come close to analyzing anything close to 500 respondents. Forrester Research came close, conducting interviews with 40 PMO leaders and executives. In their four key findings they note [12]:

- PMO leaders now have a seat at the executive table, regarded as members of executive management
- PMOs are a vital part of the strategic planning team, providing feedback about performance, labor costs and customer feedback
- PMOs build significant learning and development programs to mature project management skills
- PMOs drive success through alignment with business stakeholders and operational excellence

By far the most influential studies regarding the challenges and opportunities for the modern PMO were recently published by Gartner on the Nexus of Forces regarding the convergence and mutual reinforcement of mobile, social, cloud and information [13, 14]. This research identifies cloud computing as the glue for all the forces of the Nexus, enabling social and mobile interactions to happen at scale, and information to be freed from internal systems. A summary of PMO functions resulting from the Nexus is given in Table 4. While the 21st century functions noted in Table 3 continue to be important, more and more companies are redirecting their PMO expertise, tools and techniques at clients and external stakeholders, hoping to bolster client satisfaction with increased productivity and quality. Whether this is a good idea is debatable and only time will tell. But we must remember that the PMO isn't science, its business; rational thought doesn't always apply.

DTE Energy: A Case Study

A good example of a client-facing PMO can be found in the Project Management Institute's (PMI) selection of DTE Energy as a finalist for their 2014 PMO of the Year Award [15]. Detroit power company DTE Energy was upgrading Detroit's aging electrical infrastructure and improving service in long-neglected

1990 – 2000	2001 – 2007
• Accomplishing more work in less time with fewer resources and without any sacrifice in quality • An increase in profitability • Better control of scope changes • More efficient and effective operations • Better customer relations • Better risk identification and problem solving • An increase in quality • A reduction in power struggles • Better company decision making • An increase in business and becoming more competitive	• Standardization of operations • Company rather than silo decision making • Better capacity planning • Quicker access to higher-quality information • Elimination or reduction of company silos • More efficient and effective operations • Less need for restructuring • Fewer meetings that rob executives of valuable time • More realistic prioritization of work • Development of future leaders

Table 3. Benefits of a Project Management Office

Table 4. New Functions of a Project Management Office Gartner, 2014

2008-2014	
• Hide complexity under a layer of simplicity • Foster ideas that make technology transparent while enhancing human behavior • Empower knowledge workers to share ideas • Create lasting relationships between users and cloud-based ecosystems • Develop a discipline of innovation through information by harnessing the information in social media • Maintain flexible IT configurations	• Transition from digital marketing to digital business • Focus on consumer expectations through contextual content delivery, behavioral analysis and location awareness • Facilitate an agile response to changes and innovation in the workplace, disruptive trends, revised roles and frequent changes in providers • Plan for a future where the cloud-led model changes the nature of applications and opens digital business opportunities

areas. The PMO developed two centers of excellence, one responsible for providing resources and the other for ensuring quality. These centers provided oversight on standards, practices, coaching, mentoring and assessments, resulting in project delivery on time, on budget, with the required scope. While these are all standard early 21st century PMO functions, they also communicated outcomes to stakeholders, implemented a detailed change process to evaluate revisions to scope, budget or requirements, and facilitated the redeployment of capital as necessary. Way back in the early 2000's these functions were handled at the siloed project level. As DTE has shown though, the new nimble PMO is capable of flexible configurations to enact necessary changes and to collaborate strategically.

Confronting the Brutal facts and Reality

"Truth is incontrovertible, malice may attack it and ignorance may deride it, but, in the end, there it is." -Sir Winston Churchill

Collins explained that it is vitally important for organizations to understand the brutal facts of its environment and its problems, but to never lose faith in the organization's ability to win out in the long term. As he noted, Winston Churchill never failed to confront the most brutal facts. During WWII he created an entirely separate department outside the normal chain of command, the Statistical Office, with the principal function of feeding him—continuously and unfiltered—the most brutal facts of reality. He slept soundly knowing these facts. Recent research suggesting best organizational practices for project management similarly suggests that the best way to improve project management is to have the "difficult conversations" necessary to keep projects healthy [16]. To take this a step further, we must retain faith we will prevail in the end and confront the most brutal facts of our current reality (Collins' Stockdale Paradox).

The PMO should ensure that the brutal facts associated with a project are recognized in an effective and timely manner, so that issues and risks can be addressed and corrective action taken. Unaddressed risks and issues are compounded in a multi-project environment, and a central office for handling of various problems will ensure that personnel feel safe to report and act on their concerns.

PMO Risk Management Considerations

"You'll miss 100% of the shots you never take."
-Wayne Gretzky

- **Value**. Issues of value and the contribution or lack of contribution to project performance are key to the perceived performance and ultimately to the legitimacy of the PMO. Poor-performing PMOs are seen as contributing little to project performance, while highly valued PMOs are seen as making significant contributions to performance. The ability to show contribution to performance is critical.
- **Cost**. While many companies see fit to establish large and costly PMOs made up of a dedicated full-time team, the administrative cost of a PMO should be minimal. The key is to establish a culture wherein personnel devote their time and energy toward continuous process improvement in addition to their compensated level of effort. The PMO should be staffed with Level 5 leaders who target the success of the company over their own personal advancement. PMO membership should be considered an honor and a privilege to serve, with effort expended without additional compensation.
- **Mandate.** Organizations choose from a number of possible roles or functions when deciding on the mandate to give a PMO. They also choose between a PMO in a support role with little or no authority and a PMO with considerable decision-making power. Senior Management will need to form a balance between these roles and functions that establishes an efficient and effective PMO conducive to business culture and values.
- **Best Practices.** A Lessons Learned database is vitally important to any organization for a number of reasons. As a key mechanism for cross-project communication, it allows for sharing of best practices and fosters continuous process improvement. However, we often learn more from what we did wrong than what went right on a project (luck often enters into success). We should ensure such a database is not only useful

(i.e., searchable, organized by category, etc.), but is not viewed as a 'failure information database'. Internal marketing and communication are required to ensure this information is understood to be used for improvement and not blame.

Agile vs. the PMO

"Agile methods derive much of their agility by relying on the tacit knowledge embodied in the team, rather than writing the knowledge down in plans." -Barry Boehm

"Strange women lyin' in ponds distributin' swords is no basis for a system of government. Supreme executive power derives from a mandate from the masses, not from some farcical aquatic ceremony." -Monty Python and the Holy Grail

Agile development is characterized by frequent rapid delivery of useable software by self-organizing teams with regular adaptation to change [17]. Working software is the principal measure of progress; and increased throughput (velocity), by reduction of bottlenecks, is the primary measure of efficiency. Such methods are not very conducive to authoritarian control by the standard PMO model.

In Agile, just as in chess and in the PMO, there are multiple decisions and compromises to be made. By the second chess move there are 72,084 possible games; by the 3rd move, 9 million, and by the 4th move, 318 billion. There are more possible games of chess than there are atoms in the universe [18]. Similarly, the imposition of standard processes by a strategic PMO on Agile projects can lead to multiple responses and disastrous effects. Typically, when a PMO attempts to develop and implement a standard methodology, force common documentation archives or standardize post project reviews on an Agile project, the response is something like "Thank you very much, but we already got one, and it's very nice."

In a recent unscientific blog that I posted (there is very little research in this area), I received 72 comments on this subject from both Agile PMs and Agile Practitioners. Most PMs thought that a PMO could, if managed properly, benefit Agile teams. However, most Agile Practitioners not only viewed the PMO as the Evil Empire, but considered not only the PMO but all project managers to be waste, bottlenecks and a hindrance to velocity. Many questioned whether either should be allowed to breathe oxygen in an Agile environment.

Currently many PMOs believe that Agile is a blip on the process radar that will someday go away. Agile isn't going anywhere, mainly because customers love frequent deliveries (call it the Amazon conundrum). PMOs need to understand the Agile methods being used in their company, and manage strategic processes and decisions accordingly. Somebody's got to be the hero; it won't be the Agile Practitioner who has a manifesto and usually productivity metrics to back up their increase in productivity and client satisfaction. The successful PMO will be the one that understands the difference between a project schedule and an Agile roadmap, osmotic communication and talking, and minimal documentation vs. maximum invisible documentation. Remember that while there are more possible chess games, or decisions to be made in a technical environment, than there are atoms in the universe, if you make a mistake there is nearly an infinite amount of ways to fix it.

Conclusion

"You don't lead by pointing and telling people some place to go. You lead by going to that place and making a case." -Ken Kesey

Research has shown that PMOs are largely new creations in the corporate world and undergo frequent changes in relatively short periods [19]. Therefore it should not come as any surprise if an initial PMO has a short life span before being restructured or refocused. Effective PMOs continue to add value specifically by changing and reinventing themselves—as long as they stay focused on the principle of improving project management.

- **Establish a Strategic PMO.** A strategic PMO should be established, along with a systematic approach to create and sustain project management value through a PMO charter. There are many types of PMOs, a subject beyond the scope of this article, but aligning the goals of a PMO with the strategic direction of the company is critical to the success of this venture.
- **Create a PMO Charter.** A charter recognizes the existence of a project. The PMO should be treated as a project, with a charter that describes at a high level the business need, scope, objectives, deliverables, constraints, assumptions and approvals.
- **Use Information Radiators.** "Information Radiator" [20] is an umbrella term for a number of highly visible ways to display information, including video displays, data summaries and email newsletters. Typically used in Agile software development, it's an ideal way to inform PMO benefits and services to stakeholders and to maximize exposure of the PMO. Communicate the potential benefits early and often, supported by metrics relevant to business objectives.
- **Drive Down Decision Making.** Dr. Ostrom contended that individuals and communities could effectively manage their own collective resources without the intrusion of higher level authorities. While occasionally tactical decisions at the project level require coordination with strategic objectives of senior management, project managers are usually in the best position to make effective and timely decisions. A Strategic PMO will allow for collaborative management of project risks and issues with increased agility by the personnel closest to the action.
- **First Who…Then What**. A compelling high-level corporate vision is imperative. As Collins notes, "People are not your most important asset. The right people are." Any company should therefore concentrate on who gets on the PMO bus and what seats they take before setting a direction for the PMO.
- **Staffing with the Right People**. Collins again. While our first inclination may be to search for the most intelligent, multi-disciplined personnel to staff the PMO, we should search first for Level 5 leaders who exemplify the key trait of ambition first and foremost for the company and concern for its success rather than for one's own riches and personal renown. These

individuals should be self-effacing people with extreme humility and intense professionalism who will display the fierce resolve to do whatever is needed to make the company great.

- **Teaming: Integrating the Right People**. While Collins points out that once you have the right people in the right seats on the bus, issues such as managing change, motivating people or creating alignment largely melt away. Here I must disagree. In a perfect world where you've identified the proper people in the exact proper places I might agree. However, in our imperfect world we will need to pay special attention, especially in the initial stages, to assigning PMO members who have previously shown the ability to work together as a group.
- **Foster Innovation through Best Practices.** A recent report from Forrester Research indicates that the most successful PMOs focus on removing obstacles and delivering project management best practices to the entire company [21]. It's important to celebrate our successes and even to document them in detail for others to learn from. It's just as important, however, to focus equal emphasis on learning from those projects that appear to have run off-course [22].
- **Progress Measurement and Continuous Improvement.** As the PMO evolves, members must concentrate on the metrics by which their projects are measured as well as how process effectiveness is determined. While there must be a concerted effort to identify processes which require improvement, data collection and indicators (graphs, charts, etc.) must directly support both project and organizational goals. We can't manage what we can't measure, and unless all the projects in the company can be held up to the light and compared to each other, we have no way of managing them strategically, no way of making intelligent resource allocation decisions, no way of knowing what to delete and what to add. The PMO will assist in making these key decisions.
- **Start Small.** Tables 2, 3 and 4 provide examples of typical PMO functions in use throughout industry. There will be a strong inclination to initially load up the PMO charter with all the company's perceived problems and desired process improvements. Highly valued PMOs are seen as making significant contributions to performance and, especially in the initial stages, the ability to show contribution to performance is critical. Once the PMO is seen to produce some highly visible and useful improvements, however small, personnel will begin to value the PMO as a mechanism of positive transformation. The PMO charter can then be revised, incorporating new functions in accordance with the organization's strategic goals. Research has shown that successful PMOs start with a narrow portfolio of projects, and as project management becomes a more systematic practice, the PMOs broadened their scope. These PMOs had to demonstrate value within their first six months of existence to maintain executive support [23].
- **Consensus.** It's important to note that the PMO serves in an advisory function to the CEO. Therefore, the PMO should not necessarily seek consensus on every issue, recognizing that consensus decisions are often at odds with intelligent decisions, and that dissenting opinions may well have importance in the final analysis. The responsibility for the final decision should remain with the CEO.
- **Use Existing Assets to Assure Compliance.** Most PMOs I've studied had a tendency to make decisions and attempt to follow through on implementation on their own. Unfortunately, as an advisory committee to the CEO they had no charter to force implementation and, once forced, to determine compliance. Many organizations employ BPI and Quality Assurance teams with vast experience in implementing change and ensuring compliance. Upon CEO sign-off, the PMO should use these teams to full advantage. While maintaining a separate identity, these teams can be of great value in advancing the goals of the PMO.
- **Select PMO Functions that Make Sense for the Corporate Culture.** Although this paper lists 27 functions in common use by industry, we must recognize that most companies are unique. They make a unique product for a unique client base. While selection of PMO functions will be the responsibility of the PMO, the following functions may be considered for initial implementation:
  - > Report overall and individual project status, issues and risks to upper management
  - > Advise senior management on strategic project initiatives
  - > Provide coordination between projects, eliminate silos and ensure effective communication
  - > Provide mentoring, coaching and training in project management and methodologies
  - > Allocate resources between projects
  - > Implement and manage Best Practices and Lessons Learned databases to ensure the effectiveness of the program

Based on history and accomplishments, most companies have both great advantages and unique challenges. Continuous process improvement initiatives, along with the dedication of talented personnel, should allow most companies to continue their good to great journey, even in the current economy. The opportunity exists, however, to improve to an extent that was previously unimaginable only a few years ago. Establishing and maintaining a strategic project office will help facilitate and maintain the corporate transformation to a new level of efficiency, productivity, quality and commitment to excellence.

ABOUT THE AUTHOR

Peter D. Morris, PMP, PMI-ACP is a Process Engineering Consultant for INDUS Technology Inc. under contract to SPAWAR Systems Center (SSC) Pacific. He has worked in both the commercial and DoD/Federal sectors, including several Fortune 500 companies. He has authored numerous technical reports, including publications for the NSA, DNA, US Army NTC, USAF Space Command and the SINCGARS & JTIDS communications programs. Mr. Morris has also been instrumental in establishing successful PMOs at numerous organizations, targeting strategic Business Process Improvement associated with Project and Process Management, Engineering and Project Support enhancements.

E-mail: pmorrispmp@aol.com

REFERENCES

1. Garrett, H., "The Tragedy of the Commons." Science Vol. 162 (3859): 1243–1248. 1968.
2. Ostrom, E., Governing the Commons: The Evolution of Institutions for Collective Action, Cambridge University Press, November, 1990.
3. Boehm, B., Yang, Y., et al, Phase Distribution of Software Development, University of Southern California, 2008.
4. Kerzner, H., Strategic Planning for a Project Office, Project Management Journal, 34(2), 2003.
5. "The State of the Project Management Office (PMO) 2014, pmsolutions, 2014.
6. Booth, Rose. "IT Project Failures Costly, TechRepublic/Gartner Study Finds," TechRepublic, 2000.
7. "Gartner Says a Project Management Office Can Streamline IT Modernization", Press Release, 21 July 2008.
8. Crawford, J. Kent, "The Strategic Project Office", New York, Marcel Dekker, 2002.
9. Hill, G.M., Evolving the Project Management Office: A Competency Continuum, Information Systems Management, 21(4), 2004.
10. Hobbs, B., & Aubry, M., A Multi-Phase Research Program Investigating Project Management Offices (PMOs): The Results of Phase 1, Project Management Journal, 38(1), 2007.
11. Collins, J., Good to Great: Why Some Companies Make the Leap and Others Don't, New York, HarperCollins, 2001.
12. Strategic PMOs Play A Vital Role In Driving Business Outcomes, Forrester Research, November 2013.
13. The Nexis of Forces: Social, Mobile, Cloud and Information, Gartner Research, 14 June 2012.
14. The Nexis of Forces is Creating the Digital Business, Gartner Research, 3 December 2014.
15. Jones, T., Power Sources: 2014 PMO of the Tear Finalist, PM Network, February 2015, Volume 29, Number 2.
16. Grenny, J., Maxfield, D., & Shimberg, A., How Project Leaders Can Overcome the Crisis of Silence, Sloan Management Review, 48(4), July 2007.
17. "Principles Behind the Agile Manifesto", Agile Alliance, Beck, Kent, et al (2001).
18. Victor Allis, Searching for Solutions in Games and Artificial Intelligence, Thesis, University of Limburg, Maastricht, The Netherland, ISBN 90-900748-8-0.
19. Hobbs, B., Aubry, M., & Thuillier, D., The Project Management Office as an Organizational Innovation, International Journal of Project Management, 26.
20. Cockburn, A., Agile Software Development, Addison-Wesley Professional, October, 2001.
21. Levinson, M., "Project Management: 5 Characteristics of Transformational PMOs", CIO Magazine, April, 2011.
22. Julian, J., Cross-Project Learning and Continuous Improvement, Project Management Journal, 39(3), 2007.
23. Visitacion, M., Are You Ready To Transform Your PMO?, Forrester Research, April, 2011.

Re-Using Open Source Software in Your Software Delivery

Karen McRitchie, Galorath Incorporated
Rick Spiewak, The MITRE Corporation

Abstract. Open source software is generally available with few restrictions (depending on license) to be reused by other developers. Reuse of software presents both potential cost and schedule savings and corresponding risks to both cost and schedule. The key defining characteristic which distinguishes between risk and reward in this scenario is the quality of the software to be re-used. Well-established metrics and best practices in software development can be applied and assessed when examining open-source software for potential re-use.

Introduction

Open source software can be effectively incorporated into larger software systems. It is important to understand the origin, quality and completeness of such software. While the reuse of software can be cost-effective, it does involve cost. These costs can be taken into account using standard measurement and estimation tools based on the completeness of the package and the structure of the source code. This will help to avoid unexpected cost and schedule problems caused by incomplete or problematic source code acquired via open source. An example using the SEER for Software [1] modeling tool is used to illustrate this.

Open Source Software Defined

The definition of Open Source software is generally attributed to the Open Source Initiative [2]. In general terms, it requires the following (abbreviated) features:

- Free redistribution – no fees or royalties
- Source code (in the preferred form for modification)
- Derived works are permitted
- Limited restrictions on distribution of modified source code
- No discrimination as to persons or field of use
- License included without requiring re-licensing: not specific to a product, not restricting other software, and not technology-specific

This means that open source software is generally available to be re-used by anyone who requires the capability which it represents, and for any purpose.

Note, however, that this definition does not include any form of warranty or support for open source software. Responsibility for meeting any and all requirements, whether technical or in the area of reliability, maintainability and availability, rests with the user of open source software. This caveat also applies to any security considerations.

Licenses for open source software vary, and need to be examined to make sure that there are no unacceptable terms and conditions relative to your intended use. Take care to ensure that these do not conflict with the need to maintain modified source code without distributing it or require contributing it back to the original project if this is relevant.

Open Source Projects

There are a number of well-known, widely used open source projects. These serve as examples of the fact that open source development can produce viable, reusable products. Key instances include:

- Operating systems based on the Linux kernel (several variants)
- The Apache web server
- The Eclipse software development environment
- The NetBeans software development environment
- The Java language
- The MySQL database
- The Git version control system
- The JUnit unit testing framework for Java

These and other tools are often used to develop other open source software, which is commonly available for download or contribution on web sites such as GitHub [3] or SourceForge [4]. The overall effect is the creation of a software ecosystem, in which many developers contribute in a synergistic fashion to related projects. While this ecosystem is particularly useful to one-off or research projects which simply need a particular function in order to reach an end objective, incorporating open source into products which need to be delivered and sustained as part of a program of record requires additional consideration.

Acquiring Open Source Software

While there is no direct cost involved in acquiring open source software beyond the time and effort involved in locating and downloading it, this doesn't make it free of cost to use. There are a number of elements that factor into the ability to re-use software. These are outlined in Elements of Reusable Software, below. Missing elements are likely to require additional work including additional software development on the part of developers reusing the software.

As pointed out by Capers Jones [5], "Reuse of code, specifications, and other material is also a two-edged sword. If the materials approach zero-defect levels and are well developed, then they offer the best ROI of any known technology. But if the reused pieces are buggy and poorly developed … software reuse has the worst negative ROI of any known technology."

This means that open source software being considered for reuse should be assessed according to the standards used for

new or reused internal development by the adopting organization. Because the original developers may not be available to answer questions or provide direct support, in some respects it is more important to ensure that the initial quality meets these standards.

Additional considerations which can improve reusability include [6]:

- An active online community providing support forums
- Availability of training from authors or third parties
- Availability of paid support from authors or third parties
- Source code licensing which is appropriate for the intended use

In addition to the above attributes of open source software, it is important to distinguish the form in which this software is acquired. Some repositories, such as NuGet [7], are oriented towards ease of use and keeping current with referenced projects. This can mean providing pre-compiled components together with their dependencies. In some cases these components represent released and supported products from major companies. These are not in the same category as open source, and don't need to be treated in the same way. In cases where pre-compiled components are actually related to open source, the selection criteria may vary. If the pre-compiled components are released products these may be the appropriate choice unless there is a good reason to modify the source.

When acquiring open source software for a program of record, an approach which provides compiled components should only be used for products from an approved supplier. In this case, they should likely be downloaded separately and configuration managed in lieu of obtaining them from the repository at build time. Source repositories such as GitHub or CodePlex [8] can be used to acquire source code. Be sure to distinguish among the various types of available components and select your acquisition methods accordingly.

Applying Open Source Software

Applying open source software in a program of record requires that the development team assess the completeness and quality of the open source software under consideration using the same standards as are required for new or internally reused development. This means that the developers need to be able to incorporate the open source software into their development as if it were part of their original work.

Understanding and Inspection

It is necessary to understand open source software as well as test it to the same standards used for internal development. This should include appropriate inspection and testing using the steps suggested by Capers Jones [9] and listed in Defect Removal, below. This can serve as an important measure of the defect potential and quality of the software.

Not all of these steps can be readily applied to software acquired as open source. On the inspection side particular attention should be paid to the code, any test cases and static analysis. Automated tools are readily available to aid in this task, providing the ability to measure elements such as:

- Conformance to best practices in coding – measured by static analysis tools, generally specific to particular language(s) or development environments. Examples include Cppcheck [10] for C and C++, FindBugsTM [11] for Java and FxCop [12] or the equivalent Microsoft Visual Studio Code Analysis [13] tool for Microsoft .NET, C and C++. These tools report on potential errors in code and identify the potential consequences. The number of potential errors can be compared to standards as enumerated [14] by Capers Jones in terms of potential defect density.

While static analysis does not completely detect all types of defects, it will provide a good starting point for judging code quality. When analyzing open source, you may find it necessary to exclude pre-determined rules in areas such as spelling and naming conventions. Analysis of code which was not originally written with these conventions in mind can produce voluminous errors which will tend to mask the potentially serious defects. Another type of error which may require manual inspection to validate is a common rule against catching general exceptions. Some static analysis tools will flag this even if the exception is then processed further. Manual inspection can distinguish cases where exceptions are ignored from those whose processing doesn't conform to the analysis tool's implementation.

- Complexity analysis [15] (most commonly Cyclomatic complexity) is measured by a variety of tools, and can highlight the potential for errors due to excessive complexity.

The common rule of thumb used as a test of excessive complexity is that when Cyclomatic complexity exceeds a range of 10 to 15 that the software routine should be refactored in order to reduce this metric [16]. Other investigators have found that the probability of a routine or module being fault-prone increases dramatically starting around a measurement of 38, and approaches a near certainty at 74 and up. Note: manual inspection can mitigate this, as (for example) lengthy "switch" statements will raise the complexity measurement. By ensuring appropriate breaks, not all high-complexity routines will require re-factoring.

- A freeware application such as SourceMonitor [17], can be used to analyze source code for "quality and quantity." This tool includes calculation of Cyclomatic complexity.
- Code reviews should be performed on open source components which display either a large number of errors flagged by static analysis or high complexity numbers.

If automated unit tests are included they should be run against the acquired code. If they are not included provision should be made for developing them on at least an as-needed basis, including allocating additional resources for this purpose. For example, if a defect is found in a particular routine it is recommended that a test be written which fails (showing the defect) followed by fixing the defect and re-running the test to show success. If a particular section of code is found to be unusually prone to defects, automated unit tests should be written to exercise the public interfaces, and used to verify that incorrect results are corrected.

As the authors have shown in a previous article [18], development of automated unit tests (AUT) in place of traditional manual unit testing does not add cost during the development

cycle. However, this model doesn't apply to AUT developed after the fact. For this reason, these tests should be developed with an eye towards appropriate return on investment. In addition to the consideration above regarding defect prone code, routines whose correct operation is deemed critical to the application should be outfitted with AUT as well. One of the unheralded benefits of this type of test is that it serves as a working example of how a developer can successfully implement the underlying functions in new code, which can be a productivity enhancement.

Each of these aspects can be taken into account in modeling the potential cost of re-using the open source software under consideration.

Security Considerations

Additional inspection steps may be needed which specifically address potential security issues, classified as weaknesses or vulnerabilities [19]. This generally requires the use of specialized tools such as HP's Fortify, klocwork Insight or Coverity Code Advisor. These are static analysis tools specifically built to inspect for potential security problems. All of these support the languages most commonly found in open source software such as C++, Java and C#.

If architecture specifications and diagrams or other information is not available with open source software it may be advisable to allocate resources to generate this by using available tools which include reverse-engineering capabilities. Examples include Enterprise Architect (Sparx Systems) and Imagix 4D (Imagix Corporation).

Cost Modeling

Code reviews should be conducted on selected code samples both for the purposes of inspection and for familiarization. Developers who intend to incorporate open source software should expect to become familiar with the code in order to effectively use, test and troubleshoot systems using the open source code.

In assessing the impact of incorporating open source software, an overall estimate of resource requirements can be created by applying a model using SEER for Software.

Identify What Needs to be Costed

Open source is no different from any other estimate in that you need to have an understanding of the scope of work involved. Traditionally for software projects, this involves sizing up the software to be built and using a parametric model or productivity factors to project cost.

Open Source Cost Modeling Checklist

- Obtain code count
- Identify build configuration assumptions
- Identify source files/modules requiring manual code review. (If your estimate is being done prior to static code analysis, assume 5%-20% will require review.)
- Review results of static code analysis to identify potentially problematic modules.
- Are unit tests built-in? If, not make sure to include these as needed in your estimate
- Identify modules requiring testing
- Include size for features that need to be added or modified to meet overall requirements
- Review assumptions on productivity drivers, including experience or lack thereof for any OSS packages used
- Review allocation into roles and activities
- Assess aggregate risk to the overall effort and schedule

Cost Modeling Example

This section has an example showing the use of OpenVPN, an open source VPN package that can be used to create connections to private networks. SEER for Software will be used to model and compute effort associated with the OSS package. SEER for Software is a commercial software estimating solution that can be used for a wide range of software development and maintenance projects, published by Galorath Incorporated [20]. The general approach will be to use the OSS package size as existing code and generate assumptions related to the review/rework and testing associated with the open source.

OpenVPN

OpenVPN is a full-featured open source SSL VPN solution that accommodates a wide range of configurations, including remote access, site-to-site VPNs, Wi-Fi security, and enterprise-scale remote access solutions. Starting with the fundamental premise that complexity is the enemy of security, OpenVPN offers a cost-effective, lightweight alternative to other VPN technologies that is targeted for the small/medium business and enterprise markets [21].

Using the USC Unified Code Count (UCC) tool [22], a full count of all source files was generated for OpenVPN, as shown in Table 1 – OpenVPN SLOC Count. The vast majority are the C++ files, at over 200K Source Lines of Code (SLOC). It is important to point out that for cost modeling purposes Logical SLOC should be used and not physical lines of code which can be considerably higher.

Language Name	Number of Files	Physical LOC	Logical SLOC
Bash	10	7349	5908
C_CPP	172	62167	38789
JavaScript	1	73	48
Perl	2	47	42
Total	185	69636	44787

Table 2 – Rework Assumptions

Parameters	Function Based Sizing	Project Monitor & Control Snapshots	Labor Category Allocation	
COMPONENT: OpenVPN (Open Source)	Least	Likely	Most	Note
Pre-exists, designed for reuse	366	492	966	
Pre-existing lines of code	44,787	44,787	44,787	
Lines to be deleted in pre-exstg	0	0	0	
Redesign required	0.00%	0.00%	0.00%	
Reimplementation required	1.03%	1.03%	4.15%	Based on 33 - 133 functions requiring manual review and potential change.
Retest required	1.60%	2.40%	3.20%	10%-20% of the features/fouctions will be use and will have to go through formal testing, including test driver development

OpenVPN Cost Modeling Assumptions

- The total SLOC size will be entered as pre-existing code.
- OpenVPN supports multiple platforms and configuration, however only Windows will be supported for this implementation.
- OpenVPN will have to be validated for use and will undergo static code analysis to identify high risk modules that will require manual code review. For this example, 33 of 2020 functions had an indicated risk of High or Very High (based on Cyclomatic complexity evaluation). An additional 100 functions have indicated a medium risk. Based on this, between 33 and 133 functions (of a total 2020, or 1.63% - 6. 58%) will require manual code review and automated unit test insertion.
- In addition, static analysis was run against the C and C++ files using Cppcheck. 3 errors and 7 warnings were found. This is a low number by any measure, and indicates that the manual inspections and code reviews conducted by the developers have been reasonably effective [23].
- The application will make use of 10% - 20% of the overall features and functions and require regression test.

Because OpenVPN has not been used by the developing organization, test drivers will need to be developed.

Sizing the Rework

The above assumptions are mapped into SEER for Software inputs with the aid of the SEER for Software Rework Percentages workbook to derive rework percentages that reflect the review and test effort associated with the OSS.

Reimplementation captures the effort associated with non-automated code reviews and unit test efforts. The retest effort captures effort for running regression tests as well as developing test drivers needed for such tests. It is assumed that the overall design of the OSS will remain intact, thus no redesign. The input assumptions are expressed as a range. This is especially important when static code analysis has not been done and the extent of the review is unknown. The computed rework assumptions into the size inputs are reflected in Table 2 – Rework Assumptions:

The overall estimate is coupled with any custom or non-OSS software that needs to be developed. In this case, a component for new code for secure components was added to the estimate.

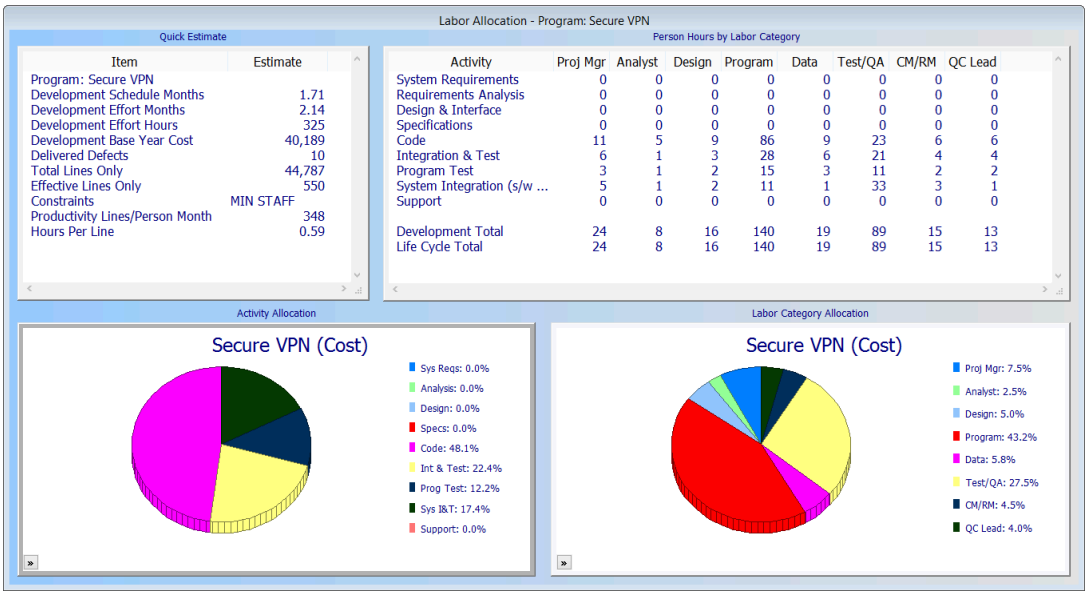


Figure 1 – Overall Estimate

Other Cost Driver Considerations

The default parameter settings are a good start for most input drivers. However some were tuned to reflect the specific situations. The following adjustments were made to default parameter settings:

- With respect to adopting the open source software, no special requirements effort is needed, so that effort component is turned off. (Requirements Definition Formality = Vlo)
- No rehosting to alternate platforms is required, the OSS packages provide multi-platform support. (Rehost from Development to Target = Nom)
- The OpenVPN has no special UI (Special Display Requirements = Nom)
- OpenVPN has added security requirements commensurate of Common Criteria EAL 1 (Security Requirements = Nom+)

The overall effort to adopt this open source packages comes to 325 hours, with a schedule of around two months (assuming effort occurs in parallel). Looking at the allocation into labor roles and activities, it is clear that this is effort is all in the code and testing phases.

While the above estimate provides a good overall planning figure, evaluation of the risks in terms of time and effort should be taken into account. Running a Monte Carlo simulation of the range of possible outcomes, it becomes clear that even at a most likely scenario, more hours and schedule should be considered, as seen in Figure 2 – Monte Carlo Simulation. In planning for contingency, using a higher confidence level such as the 70% or 80% is often prudent.

This cost modeling example focuses on the effort to adopt an open source package and should be considered part of an overall system effort. Other efforts, such as training, maintenance and deployment should also be part of a system total cost of ownership analysis.

Elements of Reusable Software

For software to be reusable by other developers, whether for modification or incorporation into larger systems, there are elements that should be included in order to reduce the cost associated with the adoption of the software. These include the following:

- Programmer's reference manual with examples for any components with public interfaces.
- Interface definitions
- List of all software components with the following information:

Purpose and function

Interfaces provided

Language/version for each module

- Complete source code:

Interface Definition Language files

Web Services Description Language files

Other source code as projects/solutions suitable for compilation/build in the Integrated Development Environment (IDE) or build/make program appropriate to the source type.

XML Schema and Schematron files

Database schema definitions as applicable

- Enterprise Architect or other Unified Modeling Language (UML) source where available
- Use cases (text and diagrams) – diagrams are included in UML design files in many cases
- Class diagrams where applicable
- Dependency diagrams if created or available
- Complete list of any third-party components with version numbers
- List of commercial and public domain software required to build the software, and the recommended order of installing these on the build machine
- Distribution package and source code of public domain components used in the software

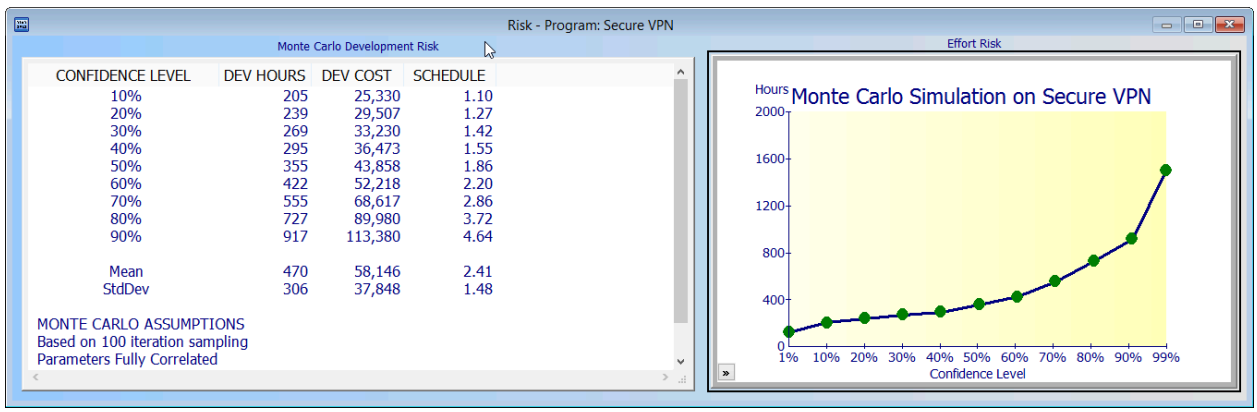


Figure 2 – Monte Carlo Simulation

- Contact information for any outside dependencies
- Build procedures, including documentation for building all components from source code

Detailed instruction on setting any necessary environment variables on the build machine, e.g., IDE options, system path.

Build procedures must be executable in a standalone development environment without requiring access to the developer's configuration management or source code control system.

No pre-installation of pre-built components included in the delivered software can be required other than any build order dependencies for components built from source as part of the build procedure.

- Test procedures – including any automated unit tests with source code, test scripts
- Installable versions of executable code, with and without debug information/symbols.
- Source for installation scripts and procedures.

While the above requirements can be specified for new or contracted development, open source software won't always include them. To the extent that these are missing or incomplete, the organization reusing the software may incur additional development or sustainment costs as a result. This must be taken into account, and decisions made as to whether to mitigate the risk associated with missing elements by assigning additional work ahead of time or to accept and account for the associated risk.

Pros and Cons of Using Open Source Software

Very often, the topic of pros and cons for open source are comparisons to using COTS software for a particular purpose. In this case, we need to consider the pros and cons in comparison to either new or continued custom software development.

Pros

- Acquisition cost: Initial acquisition cost is usually zero for source code.

Some software source code is provided free, but there is a cost for documentation and training. This is generally much lower than the cost of development.

- Maintenance: Open source software, because it is used by many others, is updated as needed. These updates are then available for incorporation. Changes you make may be adopted and incorporated into future versions. You may also benefit from changes made by others.
- Testing: Because there are many developers and users, there is ongoing testing. By participating in user forums it is possible to be alerted to potential problems ahead of encountering them in use.
- Security: Vulnerabilities are often reported to the development community, and fixes may be available in a timely fashion.

Cons

- Obsolescence: While this is a known issue with COTS software, it also exists with open source. If the system under development has a longer development and sustainment lifecycle than the open source software, it may be necessary to make custom modifications which then make it more difficult to move to a later version. Careful use of configuration management and source code control systems and good development practices can mitigate this by making it easier to identify and isolate changes. Whenever changes are contemplated, it should be determined whether a newer version should be adopted which might incorporate the needed capability, or whether the proposed changes might qualify to be submitted back to the community.
- Maintenance: While there may be many developers and users, there is not always a single point of contact for defect reporting and fixing. It may fall to the organization using the software to identify a defect, provide a potential fix, and win acceptance from the development community for implementation in a newer version.
- Quality: Coding standards may not meet those of the overall project. Static analysis may not have been applied, and peer review and inspections may not have occurred.
- Testing: The degree and depth of testing may not meet the quality standards of the adopting organization. Provision must be made for incorporating additional testing and inspection steps (see Defect Removal, below).
- Security: Secure design and security testing are not always high priorities in open source development. Depending on the security level required, standard static analysis tools may need to be supplemented with security-specific tools to examine the software. This is an additional reason to ensure that source code is used, rather than compiled versions.
- Licensing: This can vary. Some licenses require returning changes to the community. Examples of licenses applicable to cases where a requirement to post changes back is not acceptable include the Microsoft Public License, MIT license and Apache 2 License.

Defect Removal

According to Capers Jones, as cited above, combining the following recommended methods "will achieve cumulative defect removal efficiency levels in excess of 95 percent for every software project and can achieve 99 percent for some projects [24]." These are divided into categories of Pre-test and Testing. Pre-test methods consist of inspection and analysis steps. Testing is categorized according to the stage of development:

Pre-test Defect Removal

- Requirements inspection
- Architecture inspection
- Design inspection
- Code inspection
- Test case inspection
- Automated static analysis

Testing Defect Removal

- Subroutine test
- Unit test
- New function test
- Security test

- Performance test
- Usability test
- System test
- Acceptance or beta test

Open source software should be examined with respect to the needs of the application, and appropriate inspection and test steps should be performed as part of the process of incorporating this software into a resulting system. To the extent that documents or automated tests exist which represent any of these steps, the required effort to ensure appropriate quality can be reduced.

Summary

Open source software can be effectively incorporated into larger software systems. However, it is important to understand the origin, quality and completeness of such software.

While the reuse of software can be cost-effective, it does involve cost. This cost can be estimated using standard measurement tools and commercial cost estimation tools based on the completeness of the package and the structure of the source code. This will help to avoid unexpected cost and schedule problems caused by incomplete or problematic source code acquired via open source. The potential need to maintain compatibility with the original source should be taken into account as both a possible cost and a possible cost savings.

Available tools can be used to assess the quality of open source software in order to determine the likely applicability of the software to a particular system, and allocate sufficient resources to apply it effectively.

Disclaimer:

REFERENCES

1. <http://galorath.com/products/software/SEER-Software-Cost-Estimation>
2. <http://opensource.org/osd>
3. <https://github.com/explore>
4. <http://sourceforge.net/>
5. Jones, Capers. Software Engineering Best Practices: Lessons from Successful Projects in the Top Companies. McGraw-Hill, 2010. Chapter 2, p. 101
6. Discussion with David Smiley, OpenSource Connections
7. <https://www.nuget.org/>
8. <http://www.codeplex.com/>
9. Ibid, Chapter 9, pp. 618-619
10. <http://cppcheck.sourceforge.net/>
11. <http://findbugs.sourceforge.net/>
12. <http://www.microsoft.com/en-us/download/details.aspx?id=6544>
13. <https://msdn.microsoft.com/en-us/library/dd264897(v=vs.120).aspx>
14. Ibid, Table 9-8, p. 982
15. McCabe, Thomas J. "A Complexity Measure", IEEE Transactions On Software Engineering, Vol. SE-2, No.4, December 1976
16. <http://en.wikipedia.org/wiki/Cyclomatic_complexity>
17. Campwood Software, LLC - <http://www.campwoodsw.com/>
18. <http://www.crosstalkonline.org/storage/issue-archives/2008/200812/200812-Spiewak.pdf>
19. See the Common Vulnerabilities and Exposures list at <http://cve.mitre.org/>
20. <www.galorath.com>
21. <http://openvpn.net/index.php/open-source/245-community-open-source-software-overview.html>
22. <http://sunset.usc.edu/ucc_wp/>
23. Conversation with Samuli Seppänen, Community Manager, OpenVPN Technologies, Inc
24. Jones, Capers. "Methods Needed to Achieve > 99% Defect Removal Efficiency (DRE) for Software", April 9, 2014

ABOUT THE AUTHORS

Karen McRitchie is Vice President of Development at Galorath Incorporated. Ms. McRitchie is responsible for the design, development, implementation and validation of the parametric estimation relationships found in the SEER™ estimation products published by Galorath Incorporated. She has worked in all domains of cost estimation, but much of her focus has been on the software/application and information technology domains. Ms. McRitchie has participated in numerous estimation, data collection, and calibration efforts and has trained hundreds of cost analysts in the use, application, and calibration of SEER-SEM™, SEER-H™ and SEER-IT™. She has been active in the International Cost Estimating and Analysis Association (ICEAA) for many years was honored by ISPA in 2002 with the Parametrician of the Year award.

Rick Spiewak is a Lead Software Systems Engineer at The MITRE Corporation. He works as part of the Battle Management / Command and Control & Surveillance Group, concentrating on Mission Planning systems. Rick has been focusing on the software quality improvement process, and has spoken on this topic at a number of conferences as well as publishing in CrossTalk and MSDN magazines. He has been in the computer software industry for over 45 years. His experience includes developing software and managing software development for data acquisition systems, transaction processing, and data communications and networking. Rick has also taught computer architecture and data communications and networking at the graduate level. He studied quality management at Philip Crosby Associates.

Breakdown Model:

A Disruptive Software Development Lifecycle for Fault Tolerant Software Systems

Vaibhav Prakash, University of Texas
Danny Sunderesan, University of Texas

Abstract. The software development lifecycle is the most important part of Software Engineering. It determines the outcome of an idea into a tangible software. Here we present a variant of the Harmony process, the breakdown model which focuses on not only developing software but deleting all possible scenarios for failures in each phase of the development process. This framework is adaptable with existing software development lifecycles.

Introduction

Traditional software development lifecycles follow 7 core activities. They are requirements, design, construction, testing, debugging, deployment and maintenance. Naturally, apart from the requirements and testing phase, all other phases concentrate on building the software. In the requirements phase, some teams calculate the risk management which deals with the possible failure scenarios and in testing which deals with finding the loop holes based on a multitude of input values and boundary value working environments. The core idea of all software development lifecycles is to build software and not actually break it down. We believe that this is the main reason for the declining quality of software. None of the models build and destroy the software in parallel. It is quintessential to factor into our equations of how our software can fail in each phase while we are building the same. The breakdown model does exactly this—build and destroy software in parallel. Destroying software is as important as building it. Only when we understand all possible failure scenarios can we truly understand how to build software which is resistant to failure in each phase of the development lifecycle.

Methodology - Breakdown model

The normal software lifecycle architecture involves the four core parts of a software project lifecycle:

- Analysis (Requirements definition, Iterative prototypes, Object Analysis)
- Design (Architectural Design, Detailed Design)
- Implementation (Translation, Unit Testing)
- Testing (Integration testing, Validation testing, Increment Review)

The breakdown model goes a step further and adds the following addition to the process

- Analysis and Anti-Analysis
- Design and Anti-Design
- Implementation and Anti-Implementation
- Testing and Anti-Testing

What is Anti-Analysis?

In order to understand what anti-analysis is, we will first see what analysis means. Normally, the software team goes through the requirements phase and risk management is a part of it. But, in the breakdown model, a part of the team known as the anti-team (20%-25% of the team) works in breaking down the requirement documents and tries to find flaws it in. The sole purpose of the anti-team is to find ways in which the requirements definition can be proved false. The anti-analysis team can also make the requirement definition resilient to change as "changing requirements" are the number one cause for software failure

What is Anti-Design?

The same concept applies here too. A part of the team (20% - 25% of the team) acts as the anti-team here. But, the people involved in the anti-team in the anti-analysis phase cannot be duplicated here. It has to be picked from the remaining 75% of the team. The anti-design phase works in breaking down the architecture and detailed design concepts which the team have built. The anti-design team works carefully to weed out all possible scenarios where the design will fail.

What is Anti-Implementation?

A part of the team (20% - 25% of the team) acts as the anti-team here. But, the people involved in the above two phases cannot be duplicated and have to be picked from the remaining 50% of the team. The anti-implementation phase works in breaking down the implementation (such as test-driven development) while the software is being built. The anti-implementation team works in tandem with the implementation team to wipe out all possible failures in the code.

What is Anti-Testing?

The remaining team members (20% - 25%), who have not participated in the above three phases come into picture in this phase. The anti-testing team does not break the software but shows if the software works for the intended purpose. Testing and re-testing only for positive values (or) working values. They can work with the customer or simulate the intended customer who will use the software.

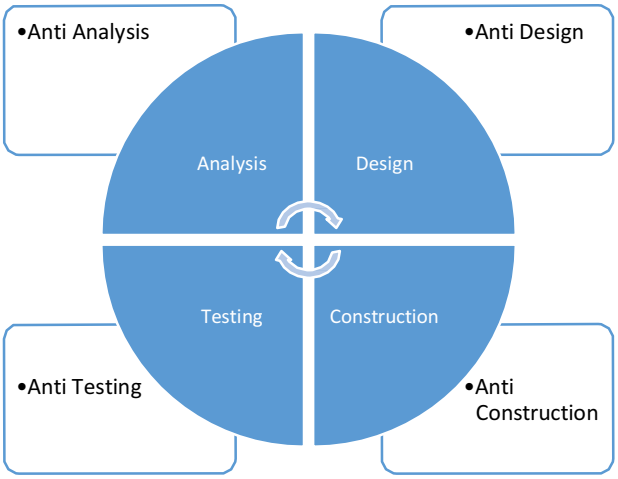


Figure 1

Number of iterations needed – N, Number of phases in life cycle – P (4 in this case)

Number of anti-team members per cycle – 100/N

Number of anti-team members in each phase – N/P

Higher the value of N, higher the probability of a software system that is fault tolerant

$N \propto$ ***Fault tolerance***

Ideally, N- ∞

Practically, N=5 should result in a fairly fault tolerant system

Figure 2

The breakdown model can be used in conjunction with the Spiral model to develop better fault tolerant systems. In order to determine the number of iterations needed for a complete fault tolerant system, we divide the number of iterations by 100, which gives the percentage of team members needed for the anti-team.

Let us take an example to better understand the above concept. If we want our software to be completed in 3 iterations, then we divide 100/3 which gives 33.3% (recurring). This means that in each iteration of the spiral, 33.3% of members act as the anti-team. Since there are primarily four phases of development, we divide it by 4, which gives 8.25% of the team to participate in anti-analysis, anti-design, anti-implementation and anti-testing separately.

Therefore, in three iterations, the entire team, in effect would have contributed to build and destroy software from end to end which gives a substantially higher probability of a fault tolerant system as all the team members would have contributed to it. The more you can involve people in the anti-teams, the better your chances are of building software which has fault tolerance.

Even when N=1(i.e. the most basic software development lifecycle incorporating the waterfall model with 4 phases viz. analysis, design, implementation and testing), the breakdown model results in a system which is 4 times more fault tolerant. This is because the system is tested only during the testing phase in the above traditional methodology. In the breakdown model, the system is broken down and tested for loop holes in each phase resulting in a better fault tolerant system.

Highlights

Weeds out errors through multiple iterations and different perspectives

We found out that with N=5. A relatively high fault tolerant system can be developed

This framework can be adapted into any of the existing software development lifecycles

Case study (Application)

We applied this to 15 software projects at the Erik Jonsson School of Engineering, The University of Texas at Dallas. All the projects were part of the coursework for graduate students. All the teams who used this framework had better fault tolerance in their software code. Although they used variants of this and incorporated the thinking into their lifecycles, it made a significant change to the product at the end compared to other teams who followed traditional lifecycles.

Conclusion

The breakdown model is best utilized when used in conjunction with the spiral or other iterative models where repeated phases are inserted into the development lifecycle. They key aspect here is using every team member's capability to see as many ways in which the system might fail in the analysis and design phase itself. The breakdown model produces better systems when used with the simplistic waterfall model too. Lastly, from the case study it is evident that the model works as intended.

ABOUT THE AUTHORS

Vaibhav Prakash is currently a Site Reliability Engineer at Microsoft Corporation in Redmond, Washington, United States of America. He holds a bachelor's degree in Computer Science and Engineering from S J College of Engineering, Mysore, India. He has completed his Master's degree in Software Engineering and Computer Networks from The University of Texas at Dallas, USA. He has published 2 papers and has done internships at The Indian Institute of Science, Bangalore, India; IBM Research and Development, Bangalore, India; Research Assistant, Multi Agent and Visualization Lab, The University of Texas at Dallas and at Microsoft Corporation, USA.

Email: vaibhav.prakash@utdallas.edu

Danny Matthew Sundaresan received his bachelor degree in Electronics and Communication from Anna University, India and is currently a Master's student in The University of Texas at Dallas, USA graduating in the field of Software Engineering. He has an experience of 6 years working as a web developer in the corporate world. He was a co founder of a freelancing web firm which brought innovative solutions to its clients during his time as an undergraduate student. His main interest involves developing user friendly tools and methods to increase the performance of the web.

Email: danny.sunderesan@utdallas.edu

REFERENCES

1. Benington, Herbert D. (1 October 1983). "Production of Large Computer Programs". IEEE Annals of the History of Computing (IEEE Educational Activities Department) 5 (4): 350–361. doi:10.1109/MAHC.1983.10102. Retrieved 2011-03-21.
2. Smith MF Software Prototyping: Adoption, Practice and Management. McGraw-Hill, London (1991).
3. Dr. Alistair Cockburn (May 2008). "Using Both Incremental and Iterative Development". STSC CrossTalk (USAF Software Technology Support Center) 21 (5): 27–30. ISSN 2160-1593. Retrieved 2011-07-20.
4. Boehm, B, "Spiral Development: Experience, , Special Report CMU/SEI-2000-SR-008, July 2000
5. Boehm, Barry (May 1988). "A Spiral Model of Software Development". IEEE Computer. Retrieved 1 July 2014.
6. Abrahamsson, P., Salo, O., Ronkainen, J., & Warsta, J. (2002). Agile Software Development Methods: Review and Analysis. VTT Publications 478.
7. Hughes, Bob and Cotterell, Mike (2006). Software Project Management, pp.283-289. McGraw Hill Education, Berkshire. ISBN 0-07-710989-9
8. <http://en.wikipedia.org/wiki/Lightweight_methodology>
9. "Crystal Methods Methodology | Infolific". Mariosalexandrou.com. Retrieved 2013-07-25.
10. "Manifesto for Agile Software Development", Agile Alliance, 2001, webpage: Manifesto-for-Agile-Software-Dev
11. Coad, P., Lefebvre, E. & De Luca, J. (1999). Java Modeling In Color With UML: Enterprise Components and Process. Prentice Hall International. (ISBN 0-13-011510-X)
12. Rosenberg, D. & Stephens, M. (2007). Use Case Driven Object Modeling with UML: Theory and Practice. Apress. (ISBN 1590597745)
13. ACM Digital Library, The chaos model and the chaos cycle, ACM SIGSOFT Software Engineering Notes, Volume 20 Issue 1, Jan. 1995
14. <http://en.wikipedia.org/wiki/Incremental_funding_methodolog y>
15. Mike Goodland; Karel Riha (20 January 1999). "History SSADM – an Introduction. Retrieved 2010-12- 17.
16. <http://www.martinfowler.com/bliki/TechnicalDebt.html>
17. Jacobson, Sten (2002-07-19). "The Rational Objectory Process - A UML-based Software Engineering Process". Rational Software Scandinavia AB. Retrieved 2014-12-17.
18. Clarus Concept of Operations. Publication No. FHWA- JPO-05-072, Federal Highway Administration (FHWA), 2005

Better Reliability Verification in Open-Source Software Using Efficient Test Cases

Patrick Pape, Mississippi State University
Drew Hamilton, Mississippi State University

Abstract. With the increasing popularity of open-source solutions in projects across varying domains and levels of dependability requirements, there is a need for a way to efficiently bring open-source software to a level that passes reliability verification testing before being integrated into a pre-existing system. The primary issues with integrating open-source software into a system is that more often than not the developmental methods cannot be verified and the software is already in a post-release version. So, how do you retain the benefits of utilizing open-source solutions to problems while bringing the open-source software to a reliable operational level that meets specifications for your project? In this article, we will discuss a method for efficiently locating key areas for the placement of error handling in order to increase fault tolerance and for drastically reducing the number of tests necessary to verify that the open-source software to be integrated meets specifications.

Introduction

Using open-source software in lieu of consumer and government off-the-shelf options for adding modular functionality to software projects or as a foundation for starting new software projects is a rapidly trending upward. [1] In our experience the government prefers an open-source software option when feasible and when robustness and security requirements of the project allow. There is abundant open-source software that performs operations across most applications. Primary drawbacks to utilizing open-source software include: no guarantee of dependability, unreliable development methods and the costs of integrating software in the late stages of development. Despite these drawbacks, open-source software is still seeing a large increase in private sector usage. [2] Open-source software can be used in many cases to reduce both the time and cost of adding new functionality or starting the development of a new project. The method discussed here will detail how to retain these cost and time savings while overcoming the reliability limitations of open-source software in systems with strict specifications for verification testing.

The inspiration for this work comes from academic research clients interested in using open-source solutions in order to save time and money in spite of software reliability concerns. The primary issue from a development point of view was trying to look at post-release software and implementing some methods for increasing the reliability of the code and performing reliability verification. One key example was an open-source mission control software for a programmable UAV. The mission control module was a mission-critical portion of the system and had strict specifications for dependability measures. The time it would take to test the system and bring it up to specifications was more than the time it would take to find alternate software that performed the same task. This dismissal of open-source software as a valid alternative to high specification systems lead to researching a way to make the method more efficient and the code more reliable.

The problem with most current solutions to the problem of making the verification process for software more efficient is that there is a lack of consideration for dealing with software only in the late stages of development and in post-release. With open-source software, unless you are involved in the development at some early stage, you are likely attempting to incorporate it into an existing system for some added functionality. This means that the solutions which focus on working on verification through each developmental iteration are not of use. A different approach must be taken in order to bring the software up to the desired level of reliable operation with acceptable overhead. Adding error handling in the late stages of development can add large overhead. The testing process discussed in this article will detail a method for making this last stage verification and reliability enhancement process more efficient.

Importance of Variables

The method utilizes static and dynamic analysis of the software to reduce the total number of tests needed to verify the software and to determine the key locations for placing error handling in order to bring the software up to the required level of reliable operation. The method focuses on utilizing the relative use of the open-source software to be integrated into an existing system to expand functionality. A recently developed metric known as importance [3] is used as a baseline for determining the priority of the variables in the software that are the highest priority for error handling. The drawbacks to this approach are a lack of cross-module measurability, meaning that each measure of importance is only relative to the other variables in a single module. The approach described here adapts the metric to establish the importance of variables across an entire system, including a large number of modules and functions.

The importance metric basically works by determining the failure rate of a variable, f, its spatial impact and its temporal impact, written as:

$$I_{v,C} = 1 \div (1 - f)^{n} \times \left(\sigma_{v,C} \div \sigma_{max} + \tau_{v,C} \div \tau_{max}\right)^{m}$$

where m and n are coefficients that can be modified to place more or less emphasis on either the failure rate of a variable or its impact on the system. [3] Failure rate in this scenario is the

frequency that a fault injected into a variable lead to a full-blown failure in the system. Spatial impact is defined as the diameter of the area in the code that is affected by the failure, written as:

$$\sigma_{v,C} = \max\{\sigma^{r}_{v,C}\}, \forall r,$$

[3] and the temporal impact is defined as the duration that a program is affected by the failure, written as:

$$\tau_{v,C} = \max\{\tau^{r}_{v,C}\}, \forall r$$

[3]. Basically, these equations state that the impacts measure the diameter of the affected area and amount of time the program state remains affected, when a variable v in component C is corrupted. A higher spatial impact indicates the difficulty of recovering from the corruption and a higher temporal impact indicates a higher chance for the program to fail.

Software Faults

The capability of utilizing an accurate measurement of the importance of a variable in a system relies on the concept of relating variables in different functions in different modules of a system to each other. The common trends of software faults and failure data from real-world case studies is explored in [4] and provides focused discussed on the localization of faults which can lead to varying types of software failures and the distribution of failures in a system. The conclusion of this study was that the primary types of faults include: requirement faults, coding faults and data problems. Of importance to the issue at hand is the conclusion that the trends of software failures are intrinsic characteristics of software faults and not specific to the individual project. This coincides with the belief that it is possible to create a metric to measuring the relative likelihood and impact of faults across any system.

Detecting and Correcting Faults

There are numerous ways to increase the reliability of a system using data flow analysis, including check-pointing [5], information flow relations [6], and other techniques. Check-pointing involves looking at the code at the instruction level and splitting it between protected and unprotected sections. In the protected code the data values are replicated and are compared at branch instructions to check for discrepancies. This concept is utilized with respect to the error handling placed into the code after the relative importance metric is calculated to ensure accurate reading and writing of variables. This error handling is currently in the form of a wrapper-based function call for each read and write of variables dependent on the level of importance. [7] The most important variables are triplicated and less important variables are duplicated during writes. This means that when the variable is being read during a program statement, the wrapper-function will check what the most common value is and return that. This allows for some data corruption without jeopardizing the values of the variable completely. Information flow and state flow analysis [8] can be used to detect errors in variables and program statements that cause undesirable actions and states in the software.

Fault-Injection Framework

There are a number of fault-injection frameworks that could be utilized depending on the application and structure of the software that is to be tested, including: PROPANE [9], MESSALINE [10] or FIAT [10]. PROPANE is an environment that supports fault-injection through mutation of source code and data errors by manipulating variables. MESSALINE [10] and FIAT [11] are used as sources of information as far as design considerations for the framework, but like PROPANE were proposed years ago and have since become less prominent solutions to new fault-injection problems. Current fault-injection frameworks are generally limited in usefulness based on the structure of the software system being tested and rely on being a part of the V&V testing process before deploying the software. These frameworks are useful for investigating the effects of individual faults on a system and identifying potentially vulnerable locations in code, but the results are not context sensitive.

The base of the method is the fault-injection framework that was written specifically to work for this particular project. The framework utilizes two models for injecting faults. The global model constrains the occurrence of the faults to dependability measurements and assumes any variable in the system could be affected by the data fault. The local model states that the types of faults to occur in the system will be transient data value faults. This means that the faults can occur at any time and any place in the software and will not remain in the program after execution stops. The framework can be split into three main components: injectors, probes, and environment simulator. The injections are done manually by inserting code into the source, depending on the type of fault different inputs are needed. For boundary testing, the desired injection value is required, otherwise the injection will target a random bit in the memory space for the variable and will flip the bit. The probe component is implemented through inserting code into the source that records the value in certain variables at different points throughout the execution of the code. The environment simulator works by emulating the existing system and controlling the execution of the target software during the intended test cases. This is done in order to get accurate results for the specific use cases of the open-source software during the testing phase.

Case Study – Mp3gain

Testing is based on an open-source program called Mp3gain which can be used to normalize volume and other audio modifications to mp3 files. Mp3gain is highly modular and was developed primarily by a single developer, a common occurrence with open-source software. The goal of the experiment was to utilize Mp3gain with several real use cases. Three test cases were selected: scan album for maximum gain adjustment, normalize volume across all tracks, and undo all changes to album. The input to the experiment was an album of 25 tracks, where faults were injected 25 times in each test at 25 equally spaced points. So, there was a single fault injection per track analysis. For each fault injection test for a variable, the program was run 100 times, meaning that each use case was run 100 times with 25 input tracks.

The first round of testing obtains the worst case impacts and determine the failure rate. The second round of testing used the probability of execution found in stage one to create a number

of fault injections relative to the actual expected number of faults to occur in that function given the intended use cases. Using the relative failure rate found in the second run of tests and the impact metrics, the local and relative importance is calculated for each variable. This process was completed a second time using another critical path of lower priority modules to show that the results would be a lower chance of causing system failure. The number of tests was tracked in order to show the difference in the total number of tests needed for completing a cost-benefit analysis on placing error handling using this method versus testing every function and variable in the software. The method was validated by comparing the total number of relevant failed test cases found versus those test cases found using local importance. The following sections will provide the results and details of the relevant stages of method.

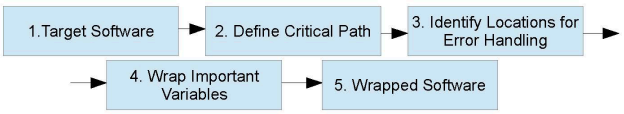


Figure 1 - Overview of the method [12]

Defining a Critical Path

The first step in the method is to determine the critical path, or the key flows of data throughout the source code. This requires that we instrument the code in order to determine which of the modules, and the functions within, are most important to the intended use case of the system. The idea is to reduce the number of unnecessary tests cases at the beginning of the testing process. The frequency that each function in each module is called during the running of the use cases is measured. This frequency value is then used along with the worst case lines of code in a function to determine the effective lines of code for each function written as:

$$\mathrm{ELoC_f} = \mathrm{LoC_{wc}} \times \mathrm{F_{cf}}$$

[12]. This measures the total number of lines of code that will be run during the execution of the program for that function. For the sake of brevity, this particular table is left to the reader to examine in [12] as it includes a large number of entries.

The next step in this stage is data flow analysis on the source to generate the caller and call graphs for each function. This is done in order to see which functions and modules are communicating with each other, so that we can track where the data is going. This is important because it helps to gauge the propagation factor of each of the functions in the code. The propagation factor is the call depth of a function divided by the maximum call depth of any of the functions, written as:

$$\mathrm{p} = \mathrm{F_{cd}} \div \mathrm{Max_{cd}}$$

[12]. Functions that have a high number of caller functions are more likely to spread corrupted data. Once these functions are given a value for priority, the priority of the modules are determined. The caller graph for the first critical path of the case study can be seen in Figure 2. The figure shows the most critical flow of data throughout the target software, where the focus of the placement of error handling will be. Each box represents a different module in the system.

A functions' priority, written as:

$$Priority = P_c \times P_e \times p$$

[12], is calculated based on the probability that a single bit fault will occur in memory related to a function, probability that the code corrupted by this fault is run in the current execution of the program, and the propagation factor. The probability of execution is written as:

$$\mathrm{P_e} = \mathrm{ELoC_f} \div \mathrm{TELoC}$$

[12], meaning the effective lines of code in the function divided by the total effective lines of code across all functions. The module priority is the summation of the priority of the functions that compose the module. A more detailed look at each stage of the method can be found in [12].

Identifying Locations for Error Handling

Once the critical path throughout the code is found, we must determine the relevant variables within the critical path to be tested. All the variables from high priority modules on the path are added. For each of these variables, a dynamic program slice is done for both the caller and called graph to get a clear measure of all the variables that are affected and affect the variable that is being analyzed. Variables with the highest relative importance ranking are located in this critical path. The output of this step is the list of all the relevant variables that gets passed to the next stage of instrumentation for fault injection testing.

Fault injection test cases are run for each relevant variable found in the critical path to obtain metrics to measure relative importance. These include: spatial impact, temporal impact, relative failure rate, and failure rate, for validation. The importance metric was taken from its generic form:

$$\mathrm{I_{v,C}} = \mathrm{G}\left[\mathrm{K}\left(\sigma_{\mathrm{v,C}}\right), \mathrm{L}\left(\tau_{\mathrm{v,C}}\right)\right]$$

in [3] and is used with the failure rate and impact of the variable to determine the importance of a variable within its own function. This equation is modified to give the importance of a variable relative to the intended use cases of the target software:

$$\mathrm{RI_{v,C}} = 1 \div (1 - \mathrm{f_r})^{\mathrm{n}} \times \left(\sigma_{\mathrm{v,C}} \div \sigma_{\mathrm{max_r}} + \tau_{\mathrm{v,C}} \div \tau_{\mathrm{max_r}}\right)^{\mathrm{m}}$$

[12] and compared to the local importance.

The results for the top threshold of variables in the case study can be seen in table 1. This shows the top fifteen percent of variables in terms of relative importance across variables in the system. Note that given how the program handled albums, the

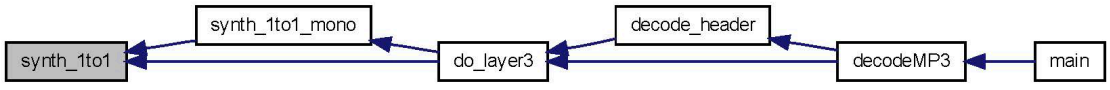


Figure 2 - Caller side for first critical path [12]

faults that manifested as failures would either cause failure for all twenty-five tracks or just one, leading to the similar temporal impact values. Another interesting note is how the curframe variable was ranked in the main module. The relative failure rate was incredibly low, but it was still ranked the sixth most important variable. This was because the fallout in the system from having this value corrupted was bad enough to outweigh the miniscule chance that the variable be read after having been corrupted.

Wrapper-Based Error Handling

An efficient and verified error handling mechanism from the same paper as the original importance metric is used. There are two stages of wrapping used in this error handling. First, any time an important variable is written to, the wrapper function is called and copies of the value are stored in case the original variable value is corrupted. The second stage is during reads of an important variable. When an important variable is read, another wrapper function is called which uses a majority voting algorithm to return the correct value. Custom thresholds are determined based on acceptable levels of overhead for determining how many variables will be wrapped. This method of wrapper was shown to be lightweight, efficient and well tested in [11].

Results

The results of the case study for the method are reassuring. The method is able to effectively reduce the total number of tests needed for reliability verification by using a measured means of removing irrelevant and unnecessary tests and focusing on only the most important variables. Table 2 shows the results of the comparison between using the given method and just the local importance. The failed tests column indicates the total number of test that resulted in system failure that occurred and the relative failed tests indicate how many of these failed tests involved variables with a significant importance rating in the system. The results show the discrepancy between the total number of test cases and how many of those test cases would have a significant impact on the system.

Table 2 also shows a clear measurement of the number of failed tests found using a modified failure rate that is relative to the whole system local measurement of failure rate. Remember that the modified failure rate is determined utilizing the probability of execution of the corrupted data. Instead of the assumption that all data corruptions are equally likely to occur, by disregarding the probability of the data fault being executed, the modified failure rate takes this into account. The number of failures that are found with the method is significantly greater than with just local module analysis.

Table 1 - Fault injection results for variables for critical path one [12]

Function	Variable	$\sigma_{v,C}$	$\tau_{v,C}$	Failure Rate	Relative Failure Rate	Relative Importance
synth_1to1	bandPtr	3	25	0.0138666667	0.0088	1.7816802003
synth_1to1	pnt	3	25	0.0088	0.0056	1.769765878
do_layer3	pcm_point	3	25	0.0284	0.0002666667	1.7511506951
synth_1to1	buf	2	25	0.0321333333	0.0206666667	1.5640653421
synth_1to1	maxAmpOnly	2	25	0.0081333333	0.0052	1.5158384332
main	curframe	2	25	0.0188	0.0000001416	1.5
synth_1to1	window	1	25	0.0734666667	0.0472	1.3772426984

Table 2 - Number of failed tests comparison [12]

Critical Path #	Failed Tests – Method	Failed Tests- Local	% difference Failure Rate	Relative Failed Tests – Local	Relative Failed Tests – Method	% difference Modified Failed Rate
CP1	2135	3647	-70.82	110	1145	90.39
CP2	6601	6551	0.76	1110	1406	21.05
CP1/2	5398	9100	-68.58	1155	1724	33.01

Table 3- Method vs. Local Cost-Benefit Analysis [12]

Mp3gain	# of tests – method	# of tests Local Cost-Benefit Analysis	% difference
CP1	47	551	91.47
CP1/2	97	551	82.40

Table 3 shows a reduced number of total tests required to identify the most important error handling locations in the open-source software. Given these initial results, the method appears to accurately locate variables with a high relative failure rate, in addition to reducing the total number of tests needed to complete this reliability verification process. Meaning that the method should decrease the time needed to verify the reliability of the target software when compared to local cost-benefit analysis, which requires that each of the variables in the system be tested in order to get an accurate reading on where to prioritize error handling mechanisms.

Conclusion

The method shows a satisfactory reduction in testing time and accurate identification of locations for error handling placement in open-source software components to be integrated into a pre-existing system. This serves as a step in the right direction for promoting a wider usage of open-source solutions in various domains. Though directed at open-source software, any post-release software with available source code could be tested using this method. The current shortcomings of similar solutions of needing to be utilized from the early stages of development and a lack of bias in the modular structure of the system do not apply for this method. The discussed method incorporates an understanding of system structure and dependability properties to give an insight into the relative operation of the system for intended use cases of the software. This allows the user of the method to focus their reliability verification testing only on the code that affects how they intend to utilize the code. The next step for this method is to further reduce the number of tests required to locate key variables for wrapping and to determine how the metrics used in the method can be expanded upon to be more accurate.

ABOUT THE AUTHORS

Dr. Patrick Pape is an Assistant Research Professor with the Distributed Analytics and Security Institute (DASI) at Mississippi State University. He holds a B.S. in Computer Engineering from the University of Alabama in Huntsville, an M.S. in Computer Science with a minor in Information Assurance and a Ph.D. in Computer Science at Auburn University. His research interests include: open-source security and reliability, test case prioritization and minimization, software fault modeling, and machine learning.

Box 9627, Mississippi State, MS 39762
Phone: (662) 325-2080
E-mail: pape@dasi.msstate.edu

Drew Hamilton is the Associate Vice President for Research at Mississippi State University and a professor of computer science and engineering. Previously he held faculty appointments at Auburn University and the US Military Academy and a visiting appointment at the US Naval Postgraduate School. Dr. Hamilton earned his doctorate in computer science from Texas A&M University. Dr. Hamilton is a distinguished graduate of the Naval War College.

Box 6343, Mississippi State, MS 39762
Phone: (662) 325-3570
E-mail: hamilton@research.msstate.edu

REFERENCES

1. Ayala, C. P., Cruzes, D. S., Hauge, O., Conradi, R. (2011). Five Facts on the Adoption of Open Source Software. Software, IEEE, 28(2), 95-99.
2. Spinellis, D., Giannikas, V. (2012). "Organizational Adoption of Open Source Software". Journal of Systems and Software, 85(3), 666-682.
3. Leeke, M., Jhumka, A. Towards Understanding the Importance of Variables in Dependable Software. In Dependable Computing Conference (EDCC). (Valencia, Spain 2010) 85-94.
4. Hamill, M., Goseva-Popstojanova, K. "Common Trends in Software Fault and Failure Data". Ed. IEEE Transactions on Software Engineering, 34 (4). 484-496. August 2009.
5. Xiong, L., tan, Q. Data Flow Error Recovery with Checkpointing and Instruction-level Fault Tolerance. In 12th International Conference on Parallel and Distributed Computing, Applications and Technologies, (Gwangju, Korea 2011). 79-85.
6. Bergeretti, J., Carre, B. "Information-Flow and Data-Flow Analysis of while-Programs". Ed. ACM Transactions on Programming Languages and Systems, 7 (1). 37-61. January 1985.
7. Leeke, M., Jhumka, A. An Automated Wrapper-based Approach to the Design of Dependable Software. In DEPEND 2011: The Fourth International Conference on Dependability. (Cta d'Azur, France 201
8. Taylor, R., Osterweil, L. "Anomaly Detection in Concurrent Software by Static Data Flow Analysis". Ed. IEEE Transactions on Software Engineering, 6 (3). 265-278. May 198043-50.
9. Hiller, M., Jhumka, A., Suri, N. PROPANE: AN Environment for Examining the Propagation of Errors in Software. In ACM SIGSOFT International Symposium on Software Testing and Analysis. (New York, USA 2002) 81-85.
10. Arlat, J., Aguera, M., Amat, L., Crouzet, Y., Fabre, J., Laprie, J., Martins, E., Powell, D. "Fault Injection for Dependability Validation: A Methodology and Some Applications". Ed. IEEE Transactions on Software Engineering, 16 (2). 166-182. February 1990.
11. Barton, J., Czeck, E., Segall, Z., Siewiorek, D. "Fault Injection Experiments Using FIAT". Ed. IEEE Transactions on Computers. 39 (4). 575-582. April 1990.
12. Pape, P. 2013. "A Methodology for Increasing the Dependability of Open Source Software Component". Master's Thesis. Auburn University, Auburn, AL.

Driving Secure Software Initiatives Using FISMA: Issues and Opportunities

Robin Gandhi, University of Nebraska at Omaha
Keesha Crosby, Tri-Guard Risk Solutions, LTD
Harvey Siy, University of Nebraska at Omaha
Sayonnha Mandal, University of Nebraska at Omaha

Abstract. Federal agencies install many security controls for Federal Information Security Management Act (FISMA) implementation. National Institute of Standards and Technology (NIST) Special Publication (SP) 800-53 revision 4 (rev4) standardizes these security and privacy controls. This article presents a study of NIST SP 800-53 security controls. The purpose is to classify the security controls from dimensions relevant to software security. This classification highlights issues and motivates opportunities to drive software security initiatives using FISMA.

Introduction

FISMA mandated security controls drive many information security programs in the federal government. But their impact on the development and/or acquisition of secure software is not well understood. Secure software (or software assurance) provides the basis for the belief that it will operate as expected in its threat environment. Such software has capabilities to resist most attacks. It can tolerate as many as possible of those attacks it cannot resist. Finally, it can contain the damage and recover to a normal level of operation as soon as possible. This article outlines a method to classify security controls based on dimensions relevant to secure software. The findings highlight issues and motivate opportunities for driving secure software initiatives using FISMA.

Let's begin by taking a look at the source of FISMA mandated security controls for a federal information system. As part of the FISMA implementation project [1], NIST has produced several key security standards and guidelines. This includes the Federal Information Processing Standard (FIPS) 199, FIPS 200, and NIST SP 800-53. Guidance within these documents work hand-in-hand for executing the first two steps in the NIST Risk Management Framework (NIST SP 800-37). Step 1 requires security categorization of information and information systems. The potential for impact to confidentiality, integrity, and availability of information determines the categorization. FIPS 199 establishes standards for categorizing information systems in this step. Step 2 requires selection of security controls based on the security categorization in step 1. In this step FIPS 200 establishes the low, moderate and high security baselines for control selection. A baseline is a set of minimum security controls defined for a low-impact, moderate-impact, or high-impact information system [2]. NIST SP 800-53 documents these control baselines as part of a larger control catalog [2].

NIST SP 800-53 specifies controls at the level of an organization or information system. There are many mandatory controls for perimeter security, system integration, operations, and organizational processes. But controls for building-security-in the information system components, i.e. software, are hard to discern. These controls are often tangled with other system concerns. In particular, organization or system level controls need expert interpretation for secure software relevance. This article addresses this issue by the development of a coding instrument. The instrument inquires the relevance of a control along the many dimensions of secure software.

The paper is organized as follows. Section 2 outlines the development of a coding instrument. Section 3 enumerates findings from applying the instrument to NIST SP 800-53. The findings provide insights that were not accessible before due to the large volume of the control catalog. They highlight software assurance practices buried within a larger organizational and information system context. To conclude, section 4 summarizes issues and opportunities identified from this study.

Coding Instrument for Software Assurance

Bootstrapping development of a coding instrument requires a recognized definition of software assurance. NIST SP 800-53 states the definition of assurance from a system perspective. Its focus is on the emergent behavior of the components for meeting the security requirements of the system. A narrower focus on software assurance exists in many definitions by government agencies (e.g. NASA, CNSS, DHS, etc.), focus groups (e.g. SAFECode) and academics/researchers. Finally, the following definition became basis of the coding instrument. CERT/SEI has also adopted this definition for their Masters in Software Assurance curriculum project.

"Software Assurance is the application of technologies and processes to achieve a required level of confidence that software systems and services function in the intended manner, are free from accidental or intentional vulnerabilities, provide security capabilities appropriate to the threat environment, and recover from intrusions and failures [3]."

Further analysis identified more dimensions. These include dimensions related to developers, software artifacts, policies, operations, weaknesses and lifecycle processes. These provide a more holistic perspective of software assurance within the coding instrument. A brief summary of the resulting 18 coding dimensions in the instrument is as follows:

The control …

… requires the application of process for software assurance: (P)

… requires the application of technology for software assurance: (T)

… requires the application of process and technology combined for

software assurance: (P+T)

…is not directly applicable to software assurance: (N)

…is withdrawn from the control catalog: (W)

The control is relevant to …

… a developer involved in the software construction or maintenance: (Developer)

… the implementation of software artifacts: (Artifact)

… software lifecycle processes: (Lifecycle)

… policies to be enforced by software: (Policy)

… software weaknesses to be avoided, accidental or intentional vulnerabilities in software or its use, or if compliance assessment with the control will fail due to a software weakness: (Weakness/Vulnerability/Failure)

… processes for software use, configuration or maintenance: (Operations)

… security capabilities to be provided by software in a threat environment: (Capability)

… software recovery from intrusions and failures: (Recovery)

The wording of NIST SP 800-53 control descriptions have direct implications on enforcement. Control refinements by interpretation in the context of software can impact enforceability. Thus, the instrument distinguishes control descriptions that are enforceable for software security. Controls which need refinement to apply in the context of software are also identified. The instrument accomplishes these tasks as follows:

The control description …

… explicitly constraints information system components, software, code, services or applications: (E)

… implicitly constrains software components, requiring expert interpretation: (I)

… as part of the supplemental guidance refers to software components, but not in the regulatory-enforced description: (IS). This considered a subset of (I).

Each security control beings with the phrase "the organization" or "the information system." These terms are defined as: "… information system refers to those functions that generally involve the implementation of information technology (e.g., hardware, software, and firmware). Conversely, the term organization refers to activities that are generally process- driven or entity-driven—that is, the security control is generally implemented through human or procedural-based actions. Security controls that use the term organization may still require some degree of automation to be fulfilled." [2] This characteristic of a control description is captured as follows:

The control description…

… starts with the phrase "The organization": (ORG)

… starts with the phrase "The information system" (SYS)

The authors applied the final instrument to investigate each NIST SP 800-53 security control. This includes controls in 26 families, including the new privacy families. A total of 958 security controls, including control enhancements, were part of the study.

To begin the study, the four authors of this article reviewed each control. Later in a group session the authors discussed controls with divergent categorizations. The authors performed peer evaluations of early coding efforts to ensure consistent instrument use. These peer evaluations helped identify and remove sources of ambiguity early in the process.

The process resulted in a preliminary list of software assurance related controls. For feedback the authors disseminated these controls using the NIST software assurance mailing list. Several community members provided feedback. Based on the feedback and internal team review, the authors added 17 controls to the initial set of 535 controls. This brought the total number of software assurance related controls to 552. This list of security controls can is available here [4].

Observations and Findings

This section reports observations from applying the instrument to NIST SP 800-53 security controls. A brief discussion of significance follows each observation.

Do NIST SP 800-53 controls emphasize software assurance related topics?

- Target of observation: # of controls and control families relevant for software assurance
- Observed data:
  - 57.62% (552/958) controls are relevant to software assurance
  - 69.23% (18/26) families have controls relevant to software assurance
- Significance: Software is a key element in a majority of information system components. The instrument observed relevance for software assurance across NIST SP 800-53 security controls.

How obvious are the control interpretations for software security?

- Target of observation: # of Explicit and Implicit controls
- Observed data:
  - 189 controls are Explicit
  - 363 controls are Implicit
- Significance: A large number of implicit controls show a significant burden for stakeholders in the A&A activities. Stakeholders need to interpret, negotiate and agree upon implicit software assurance related controls.

How many controls related to software security are strictly enforceable?

- Target of observation: # of Explicit controls assigned to baselines
- Observed data:
  - 16 explicit controls are assigned to the LOW baseline.
  - 38 explicit controls are assigned the MODERATE baseline.
  - 53 explicit controls are assigned the HIGH baseline.
  - 342 controls (62%) are not assigned to any baseline.
- Significance: A small percentage of software assurance related controls are enforceable by minimum security baselines. Furthermore, a large number of software assurance controls remain unassigned to any baseline. The A&A activities rely on the effectiveness of the tailoring step to select the unassigned controls. Tailoring activities adjust the control baselines to a level commensurate with the perceived risk.

Which control families have a high density of explicit software assurance controls?

- Target of observation: % of software assurance controls with the Explicit (E) coding dimension within a control family
- Observed data:
  - Percentage of software assurance controls with the

Explicit (E) coding dimension within control families. See figure 1.

- Significance: Figure 1 shows that explicit controls are concentrated in the SA, SI, SC and CM families. This observation aligns well will the assertions made by NIST SP 800-53 authors. They have described SA, SI and SC control families with the most emphasis on software assurance. There is one surprising observation. The AC family has the least amount of explicit controls. But this family also has the most number of software assurance controls (95 software assurance controls).

What topics do explicit software assurance controls focus on?

- Target of observation: Co-occurrence of Developer, Artifact, Lifecycle, Policy, Weakness/Vulnerability/Failure, Operations, Capability and Recovery coding dimensions with the Explicit coding dimension.
- Observed data:
  - 61.38% Operations
  - 47.09% Capability
  - 39.68% Artifact
  - 36.51% Developer
  - 32.80% Lifecycle
  - 20.11% Weak/Vuln/Fail
  - 15.87% Policy
  - 7.94% Recovery
- Significance: The observed data shows that explicit control descriptions are not balanced. NIST SP 800-53 authors bias them more towards software use, configuration, maintenance and functional capabilities. Guidelines for developers, software artifacts and lifecycle activities form the next set of biases. Finally, weakness/vulnerability/failure topics seem to get much less mention compared to other topics.

Which control families have the most number of software assurance controls? What software assurance topics do they cover?

- Target of observation:
  - Within a control family
    - # of software assurance related controls within a family (includes both explicit and implicit controls)
  - frequency of occurrence for Developer, Artifact, Lifecycle, Policy, Weakness/Vulnerability/Failure, Operations, Capability and Recovery coding dimensions within a family as well as across families
- Observed data:
  - # of software assurance related controls within a family as shown in Figure 2
- Frequency of occurrence for Developer, Artifact, Lifecycle, Policy, Weakness/Vulnerability/Failure, Operations, Capability and Recovery coding dimensions in a control family. Top three coding dimensions in the top five control families with the most number of software assurance controls are shown in Figure 3.
- Significance: The most number of software assurance related controls come from the AC family. This family also has a high percentage of implicit controls. Like this family, other control families also exhibit tangled and hidden software assurance concerns.

The next observations show that Operations and Capability dimensions dominate several control families. But, the other co-occurring dimensions do reflect the focus of the family. For example, Artifact, Developer and Lifecycle dimensions best characterize the SA family. Policy dimension best characterizes AC and IA family. Finally, the Weakness/Vulnerability/Failure dimension best characterizes SI family.

The Developer dimension frequently overlaps with Artifacts, Lifecycle, Operations dimensions. This is because process-related activities (Lifecycle and Operations) involve developers. They also produce software artifacts that have to meet certain standards.

Next, there are many more controls coded as Operations (337) than Lifecycle (113). This implies that NIST SP 800-53 controls focus more on operational issues than on the development process. Roughly half of the controls coded as Lifecycle are also coded as Operations (51/113). This implies that many of the lifecycle processes are also biased towards operations.

Finally, there are many more controls coded as Capability (319) than Artifact (151). This implies that NIST SP 800-53 controls emphasize functional security requirements a lot. But focus less on placing requirements on the artifact creation process.

How do "organization" related controls compare to "information system" related controls for software assurance?

- Target of observation: Co-occurrence of ORG and SYS

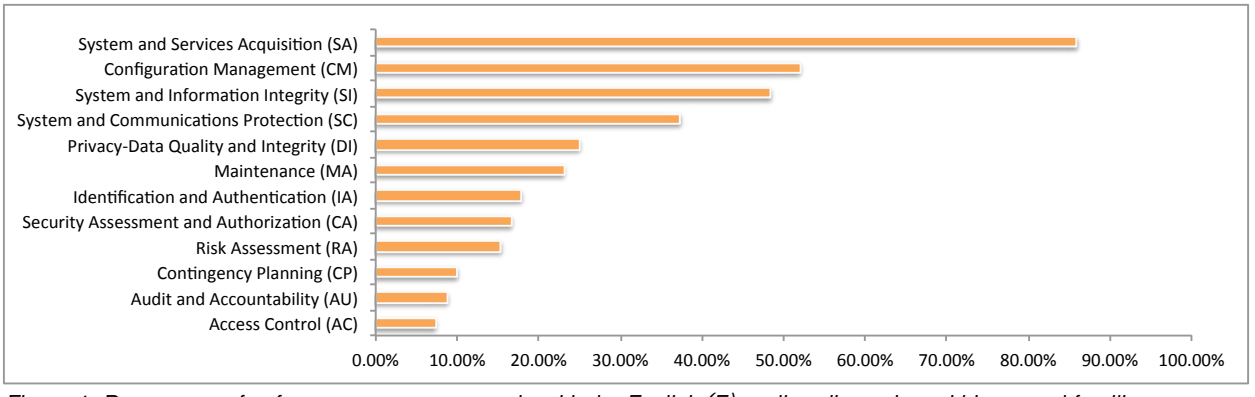


Figure 1: Percentage of software assurance controls with the Explicit (E) coding dimension within control families

Figure 2: Total # of software assurance controls across all families

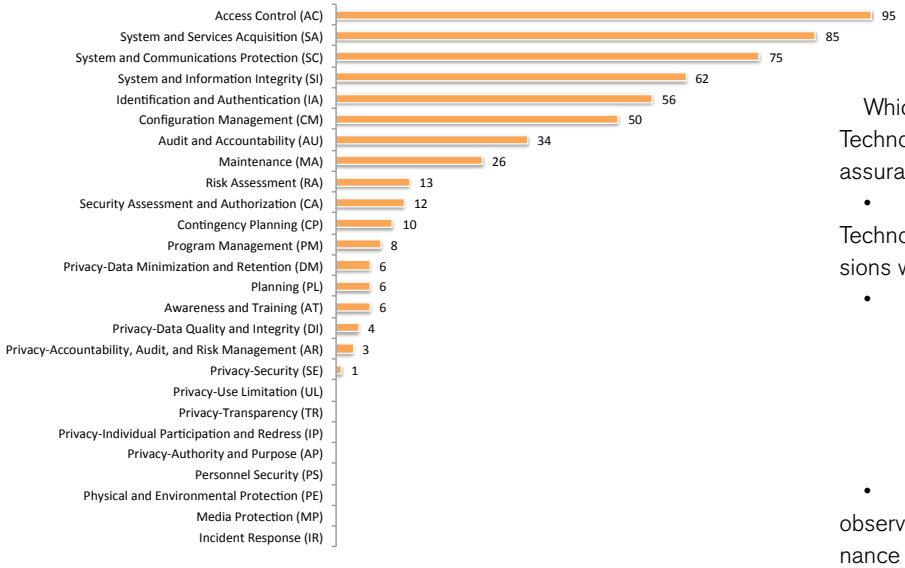
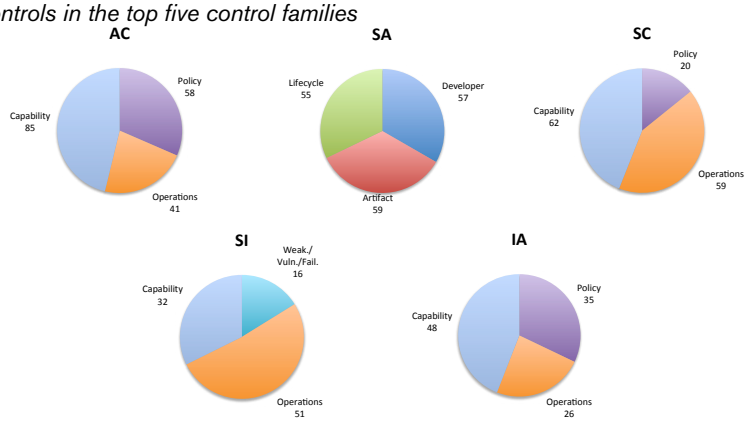


Figure 3: Distribution of Developer, Artifact, Lifecycle, Policy, Weakness/Vulnerability/Failure, Operations, Capability and Recovery coding dimensions for controls in the top five control families



coding dimensions with respect to Process (P), Technology (T), (P+T), and Explicit (E) coding dimensions

- Observed data:
  - 93.62% (191/204) of SYS controls are purely Technology-oriented
  - 99.7% (347/348) of ORG controls are Process-oriented,
  - 51.14% (178/348) of ORG controls have both a Process and Technology component (P+T).
  - ~45% (154/348) of ORG controls are Explicit
  - ~17% (35/204) of SYS controls are Explicit.
- Significance: This observation confirms that SYS controls suggest technological solutions. Next, almost all ORG controls are process oriented. But more than half of these controls also have a technological component. These observations align well with the definition of SYS and ORG in NIST SP 800-53.

Only a small percentage of explicit controls are SYS compared to ORG. This suggests that explicit controls recommend process and technology combined solutions over just technology.

Which control families predominantly emphasize Process, Technology or Process and Technology combined for software assurance?

- Target of observation: % of controls with Process (P), Technology (T) or Process and Technology (P+T) coding dimensions within a control family
- Observed data:
  - Over 50% Process (P): MA (19/26), SA (55/85)
  - Over 50% Technology (T): AC (67/95), AU (20/34), IA (29/56), SC (41/75)
  - Over 50% Process and Technology (P+T): CA (10/12), CM (28/50)
- Significance: The following justifications explain these observations. Process based controls are common for maintenance and acquisition activities. Emphasis on process in MA and SA control families reflects this. Next, automatic control mechanisms are common for access control, audit, identification, authentication and secure communications. Emphasis on technology in AC, AU, IA and SC control families reflects this. Finally, automated mechanisms often support manual processes of performing security assessments. Emphasis on process and technology combined in CA and CM families reflects this.

Issues and Opportunities

Information systems are software intensive. As a result, weaknesses in software components present a significant source of risk. Due to many implicit software assurance controls these risks are not well understood in the context of FISMA. This makes it difficult to manage software assurance as a first-class entity in the system lifecycle. Furthermore, in new system acquisitions, stakeholders tailor security controls based on system needs. During tailoring, stakeholders address many system security controls. But software security gets less attention than it deserves. Thus, the system matures with unmitigated software deficiencies and flaws. At the same time, software evolves with new features and capabilities more rapidly than the system. This fact is also evident during security test and evaluation as well as operational test and evaluation. During these assessments software components are many development versions ahead of the system maturity.

The Common Weakness Enumeration (CWE) provides a unified and measurable set of weaknesses. But, this large enumeration of over 700 weaknesses presents a significant cognitive challenge. A&A stakeholders need to understand what weaknesses are most relevant to security controls. Yet many security controls do not provide any CWE selection guidance. For example, supplemental guidance for the SI-2 Flaw Remediation control states the following. "...Organizations take advantage of available resources such as the Common Weakness Enumeration (CWE) or Common Vulnerabilities and Exposures (CVE) databases in remediating flaws discovered in organizational information systems." It is encouraging to see the mention of standard enumerations of software weakness like CWE. But, assessing compliance with this control will likely not be repeatable or uniform.

Finally, controls to avoid some of the most egregious software weaknesses do exist in the catalog. For example, controls for static code analysis, threat and vulnerability analyses exist. But they are not assigned to any control baseline (not even high-impact!). NIST SP 800-53 categorizes them as assurance controls. Stakeholders can use these as needed in different situations but not mandated. So it becomes easy for a software developer or organization to just "tailor these controls out." Tools that support security A&A activities make it even easier to filter out these controls.

While there are issues, as with any other A&A process, many opportunities also exist. NIST SP 800-53 rev 4 control catalog has a comprehensive set of requirements to develop or procure secure software. But the FIPS 200 minimum security baselines needs to include them in the low, moderate, or high-impact baselines. To build-security-in, the bar needs to be raised.

NIST SP 800-53 controls are often specified independent of specific technologies and platforms. As a result, the controls align well with abstract "Class" and "Base" level software weaknesses in the CWE. Thus, developing mappings between security controls and standard software weaknesses is essential. This effort is currently being undertaken using assurance cases as a mapping mechanism. These results are beyond the scope of this article. Finally, many relationships exist among controls as well as among CWEs. These will be essential to unravel the cascading dependencies among system components.

Just regulatory processes and controls alone cannot guarantee secure software. But, they do play a significant role in making software security programs a strategic priority. Our goal is to make software assurance related controls easily understood, communicable, and manageable. These attributes are an essential precursor to measure control effectiveness for software security. That is if the controls do in fact lead to reduction in security weaknesses in software. Also, prove the impact of these controls for acquiring software that can be securely configured, deployed and maintained.

Acknowledgements

This research is partially funded by the US Department of Homeland Security Science and Technology Directorate Cyber Security Division, D13PCC00228, "An Engagement to Look Forward to: Security Requirements and Software Weaknesses." We also thank the anonymous reviewers for their thoughtful comments on previous drafts of this paper.

ABOUT THE AUTHORS

Robin A. Gandhi, Ph.D. is an Associate Professor in the College of Information Science and Technology at the University of Nebraska, Omaha. He received his Ph.D. from The University of North Carolina at Charlotte. The goal of Dr. Gandhi's research is to develop theories and tools for designing dependable software systems that address both quality and assurance needs. He is a member of DHS Software and Supply Chain Assurance Working Group on Workforce Training and Education.

Keesha M. Crosby is the Founder and CEO of Tri-Guard Risk Solutions, Ltd (T-GRS). Prior to founding T-GRS, Crosby served as industry and government subject matter expert in software assurance as well as system security engineering arena. She has authored articles for IEEE and several patents pending. T-GRS has a tool called SACRE (Software Assurance Compliance verification and Risk Evaluation) which automates the decision making for developers and auditors based on weaknesses likelihood of breach.

Harvey Siy, Ph.D., is an associate professor in the Department of Computer Science at the UNO. He received his doctorate in computer science from the University of Maryland at College Park. He conducts empirical research in software engineering to understand and improve technologies that support the development and evolution of reliable software-intensive systems. Siy has previously held positions at Lucent Technologies and its research division, Bell Laboratories.

Sayonnha Mandal is currently pursuing her Ph.D. in Information Technology at the University of Nebraska, Omaha.

REFERENCES

1. "FISMA Implementation Project." NIST Computer Security Division. N.p., n.d. Web. 28 May 2014. <http://csrc.nist.gov/groups/SMA/fisma/index.html>.
2. United States. National Institute of Standards and Technology. SP 800-53 Revision 4. Security and Privacy Controls for Federal Information Systems and Organizations. N.p., Apr. 2013. Web. <http://csrc.nist.gov/publications/PubsSPs.html#800-53>.
3. Mead, N., Allen, J., Ardis, M., Hilburn, T., Kornecki, A., Linger, R., & McDonald, J. (2010). Software Assurance Curriculum Project Volume I: Master of Software Assurance Reference Curriculum (CMU/SEI-2010-TR-005). Retrieved November 07, 2013, from the Software Engineering Institute, Carnegie Mellon University website: <http://resources.sei.cmu.edu/library/asset-view.cfm?AssetID=9415>
4. Draft list of software assurance related NIST SP 800-53 rev4 controls: <http://faculty.ist.unomaha.edu/rgandhi/swa/controls.pdf> <http://resources.sei.cmu.edu/library/asset-view.cfm?AssetID=9415>

Upcoming Events

Visit <http://www.crosstalkonline.org/events> for an up-to-date list of events.

13th Annual IEEE Consumer Communications & Networking Conference
Las Vegas, NV
January 9-12, 2016
http://ccnc2016.ieee-ccnc.org/

12th Annual Open Forum for Large-Scale Network Defense Analytics
Daytona Beach, FL
January 11-14, 2016
http://www.cert.org/flocon/

International Conference on Verification, Model Checking, and Abstract Interpretation 2016
St. Petersburg, FL
January 17-19, 2016
http://conf.researchr.org/home/VMCAI-2016

ICCMS 2016: the 7th International conference on Computer Modeling and Simulation
Brisbane, Australia
Jan 18-19, 2016
http://www.iccms.org/

POPL 2016 Annual Symposium on Principles of Programming Languages
January 20-January 22, 2016
St. Petersburg, FL
https://regmaster4.com/2016conf/POPL16/register.php

D2D 2016: the First International Workshop on Data to Decision
February 3-5, 2016
Laguna Hills, CA
http://ssrg.nicta.com.au/Events/conferences/D2D2016/

Developer Week
February 12-18, 2016
San Francisco, CA
http://www.developerweek.com/

MODELSWARD 2016- The 4th International Conference on Model-Driven Engineering and Software
Rome, Italy
Feb 19-21, 2016
http://www.modelsward.org/

The Eleventh International Conference on Digital Telecommunications (ICDT 2016)
February 21-25, 2016
Lisbon, Portugal
http://www.iaria.org/conferences2016/ICDT16.html

14th USENIX Conference on File and Storage Technologies
February 22-February 25, 2016
Santa Clara, CA
https://www.usenix.org/conference/fast16

The 9th ACM International Conference on Web Search and Data Mining
February 22-February 25, 2016
San Francisco, CA
http://www.wsdm-conference.org/2016/

LEAN AND SIX SIGMA CONFERENCE
February 29- March 1, 2016
Phoenix, AZ
http://asq.org/conferences/six-sigma/

SIGCSE 2016
March 2-March 5, 2016
Memphis, Tennessee
http://sigcse2016.sigcse.org/

IoT Dev + Test Conference
April 17-22, 2016
San Diego, CA
https://iotdevtest.techwell.com/

People-less Requirements and Analysis

The story goes that that a general building contractor was looking to cut expenses, and noticed that he had two bricklayers working on the same project. He decided that he could get rid of one, and still finish the project on time. He decided to ask each of the two bricklayers what they though of their jobs.

He asked the first bricklayer "What do you do every day?" The first bricklayer replied "Every morning, I can't wait to get up and come into work, inspired by what I will get to accomplish that day. I prepare my bricks and mortar, and work on raising a cathedral to the sky. My humble bricklaying will help finish this work of art, and the glory and majesty that the cathedral presents will be due, in some small measure, to the quality of my work!" The general contractor, moved beyond words, wiped a tear from his eye, and sought out the second bricklayer.

He asked the second bricklayer the same question - "What do you do every day?" The second bricklayer had a totally different attitude. The bricklayer replied "I come in exactly at 8, no earlier. I mix my mortar, and – except for two breaks and lunch, I pile one stupid brick on top of another. I can't wait for the 5 p.m. bell, so I can clean my trowel and mortar bucket and go home."

The general contractor realized the choice was obvious. Without hesitation, he quickly decided to fire the first bricklayer. You see, the two bricklayers were supposed to be building a small utility shed, not a cathedral.

Back in 1976, I was a young applications programmer working at Offutt AFB. Part of my job involved supporting the programs for handling collection and analysis of satellite data. A Lt. Col, who was one of the more experienced analysts, asked me to write a special program for him to help reduce some data. I don't think the term "data analytics" existed yet – but that was what we were accomplishing. To expedite the program – I wanted to mix the data into a common file and store it on a tape drive (1975. Honeywell 6800. 96K of main memory. 4 tape drives. Card input. What more could you ask for?)

Looking for a way to distinguish one set of data from another, I realized that I would need a record separator to help me analyze the data. I asked the user if "special characters" were part of the input data – and was informed that no special characters were ever used.

Armed with this data, and a copy of Knuth's "The Art of Computer Programming: Volume 3 Sorting and Searching," I started writing code. Because the actual data was classified, I did not have access to the actual input data yet – an unclassified system was used for development and testing. Given a schema (a description of the physical format of the eventual input file), I created a series of dummy records to test my program. In short time, I had a working prototype. Several days of testing confirmed my obvious skill and both a designed and developer. It was efficient, concise, and gave accurate results. My program was ready for real data.

The trial run of my masterpiece was scheduled during the active database downtime, or what we called "night processing." During the day, the analysts needed the database "live", so background processing that modified or manipulated the data ran every night. My program was scheduled for 2 a.m. At 2:01 a.m., I was woken up from a sound sleep to hear an operator tell me that my program has crashed, and in fact had crashed immediately upon starting execution. Not much else he could tell me (remember that classified part?) so I got dressed and headed into the computer room. At about 3 a.m., I was examining the 96K core dump to find the status of registers, files, and program counters (remember the Honeywell 6800? 96K of an octal dump.)

It took quite a while to decipher the dump, but I eventually discovered that the first character of the first input file was a "!". As a matter of fact, the entire classified input file was littered with "!", "#", "#", and every other special character you could imagine.

By this time, it was 6 a.m. – so I just hung around for the Lt. Col to show up. When he finally arrived at his desk, I showed him the input file and the program dump, and reminded him that he was told me that there were no special characters in the input file.

His response? "Exclamation marks? Those aren't special characters. We use them all the time!"

And the moral of the story is that I was a young and inexperienced programmer, who should have known to examine the input files themselves, rather than just a schema of the file.

Or maybe the moral is that users and developers (and analysts and testers and maintainers) all speak a different language – and the same word carries different connotations and meanings for each person.

The hardest part of building large software? Communications. Talking to all the users, and fathering their requirements. Determining what is a "requirement" and what is just a "that would be nice to have, but we could live without it." Determining from the user how to test each requirement. Then explaining to the users how to correctly run the system, including how to handle occasional problems, shortcomings, and failures.

Come to think of it – compared to working with lots of people, coding is probably the easy part.

David A. Cook
Professor of Computer Science
Stephen F. Austin State University
cookda@sfasu.edu

CrossTalk / 517 SMXS MXDED
6022 Fir Ave.
BLDG 1238
Hill AFB, UT 84056-5820

PRSRT STD
U.S. POSTAGE PAID
Albuquerque, NM
Permit 737

CrossTalk thanks the above organizations for providing their support.