CROSSTALK

September 2003 **The Journal of Defense Software Engineering** Vol. 16 No. 9

defect management

Policies, News, and Updates

4 **2003 U. S. Government's Top 5 Quality Software Projects**
Nominations have begun for your organization to enter to become a Top 5 winner.

Defect Management

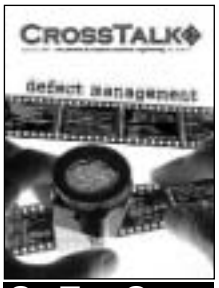
5 **The Bug Life Cycle**
Want to improve testing and quality assurance efforts? These authors discuss careful handling and tracking of software bugs by using a database and a little organization.
by Lisa Anderson and Brenda Francis

9 **Comparing Lean Six Sigma to the Capability Maturity Model**
A comparison of differences and commonalties between these two process improvement efforts shows that adding Lean Six Sigma to the mix can further reduce software defects.
by Dr. Kenneth D. Shere

13 **Managing Software Defects in an Object-Oriented Environment**
This author presents a fault model and describes steps and opportunities to detect and remove defects in an object-oriented environment.
by Houman Younessi

17 **Defect Management Through the Personal Software Process**
These authors quantitatively demonstrate defect reduction using the Personal Software Process along with a substantial reduction in test time.
by Iraj Hirmanpour and Joe Schofield

21 **Defect Management in an Agile Development Environment**
This article discusses a set of best practices associated with the Agile+ methodology that focuses on preventing both requirements and implementation defects.
by Don Opperthauser



ON THE COVER

Cover Design by Kent Bingham. NOTE: The code on the front cover under the loupe is actual C code that won the 1984 "International Obfuscated C Code Contest." It simply prints out "Hello, World!" See <www.ioccc.org>.

Software Engineering Technology

25 **Lessons Learned From Another Failed Software Contract**
An examination of the failure of a major avionics modernization program reveals a list of timeless development and management issues that continue today to derail software development projects.
by Dr. Randall W. Jensen

Open Forum

28 **Defect Management: A Study in Contradictions**
Management's full support of defect reporting and documentation is key to utilizing the findings of defect management in all phases of software development and maintenance.
by Raymond Grossman

Departments

3 From the Publisher

12 Coming Events

20 STC 2004 Call for Speakers/Exhibitors

24 Web Sites

31 BackTalk

CrossTalk

SPONSOR	*Lt. Col. Glenn A. Palmer*
PUBLISHER	*Tracy Stauder*
ASSOCIATE PUBLISHER	*Elizabeth Starrett*
MANAGING EDITOR	*Pamela S. Bowers*
ASSOCIATE EDITOR	*Chelene Fortier-Lozancich*
ARTICLE COORDINATOR	*Nicole Kentta*
CREATIVE SERVICES COORDINATOR	*Janna Kay Jensen*
PHONE	(801) 586-0095
FAX	(801) 777-8069
E-MAIL	crosstalk.staff@hill.af.mil
CROSSTALK ONLINE	www.stsc.hill.af.mil/crosstalk
CRSIP ONLINE	www.crsip.hill.af.mil

Subscriptions: Send correspondence concerning subscriptions and changes of address to the following address. You may e-mail or use the form on p. 24.

Ogden ALC/MASE
6022 Fir Ave.
Bldg. 1238
Hill AFB, UT 84056-5820

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.stsc.hill.af.mil/crosstalk/xtlkguid.pdf>. CROSSTALK does not pay for submissions. Articles published in CROSSTALK remain the property of the authors and may be submitted to other publications.

Reprints and Permissions: Requests for reprints must be requested from the author or the copyright holder. Please coordinate your request with CROSSTALK.

Trademarks and Endorsements: This DoD journal is an authorized publication for members of the Department of Defense. Contents of CROSSTALK are not necessarily the official views of, or endorsed by, the government, the Department of Defense, or the Software Technology Support Center. All product names referenced in this issue are trademarks of their companies.

Coming Events: We often list conferences, seminars, symposiums, etc. that are of interest to our readers. There is no fee for this service, but we must receive the information at least 90 days before registration. Send an announcement to the CROSSTALK Editorial Department.

STSC Online Services: www.stsc.hill.af.mil
Call (801) 777-7026, e-mail: randy.schreifels@hill.af.mil

Back Issues Available: The STSC sometimes has extra copies of back issues of CROSSTALK available free of charge.

The Software Technology Support Center was established at Ogden Air Logistics Center (AFMC) by Headquarters U.S. Air Force to help Air Force software organizations identify, evaluate, and adopt technologies to improve the quality of their software products, efficiency in producing them, and their ability to accurately predict the cost and schedule of their delivery.

Managing Defects Together

Many software managers and practitioners consider peer reviews to be a principal defect management technique. The purpose of disciplined peer reviews is to find and remove defects early, which in turn will reduce rework later. Peer reviews can take place during all phases of a software project life cycle and can be a review of any work product. Planning documents, schedules, requirements specifications, interface documents, test plans, and software code are examples of work products that can be peer reviewed. Although this issue of CrossTalk is far from a piece of code, it is a work product and has been subject to a formal review process similar to a peer review.

For those of you who have never had the opportunity to participate in a disciplined peer review, here is some insight from my peer review experiences while on a software development team. In most cases, I was given paperwork (code) to review approximately a week ahead of the scheduled peer review meetings. I found it most comforting that the focus of the peer review meeting was on the work products and not the software developers. We used checklists to guide us in finding defects.

My roles in the peer reviews varied from leader, recorder, and reviewer to author. As a code-writer (or author), the meetings were a great help to me as work product defects were uncovered in a nonconfrontational team setting. The roles and responsibilities of the review members, as well as the meeting goals, were well understood by all. This helped make the review process routine and a time for team support. Through my peer review experiences, I gained an invaluable understanding for project teams working together to meet the end goal of producing a quality product.

As we focus this month's issue on defect management, we begin with *The Bug Life Cycle* by Lisa Anderson and Brenda Francis. This article emphasizes the need for teams to carefully handle and track software bugs throughout a project's life cycle. Defect tracking through a database and enforcing policies and procedures are some of the methods suggested to improve testing and quality assurance efforts.

Next, two approaches to acquiring and developing quality software are compared by Dr. Kenneth D. Shere in his article *Comparing Lean Six Sigma to the Capability Maturity Model*. Shere discusses the primary differences and common goals between these process improvement approaches aimed at reducing software defects. Our issue continues with a look at defect management when utilizing object-oriented approaches. In *Managing Software Defects in an Object-Oriented Environment*, Houman Younessi presents a fault model and describes the many steps and opportunities to detect and remove defects in an object-oriented environment.

The Software Engineering Institute's Personal Software Process℠ (PSP℠) is yet another disciplined method aimed at managing and reducing defects. Iraj Hirmanpour and Joe Schofield share their experience with the PSP defect management framework in *Defect Management Through the Personal Software Process*. They provide good insight into how the use of defect collection and analysis metrics can benefit organizations engaged in software process improvement.

In *Defect Management in an Agile Development Environment* by Don Opperthauser, we are reminded of the importance of finding defects in the requirements phase of software projects. Opperthauser discusses a set of best practices associated with the Agile+ methodology that focuses on the prevention of requirements and implementation defects. In *Lessons Learned From Another Failed Software Contract*, Dr. Randall W. Jensen's examination of a failed major avionics modernization program uncovers issues that continue to derail software development projects.

In our Open Forum section this month, Raymond Grossman brings us *Defect Management: A Study in Contradictions*. Grossman discusses how the utility of defects can be diminished throughout the software development process if management is not involved and supportive of defect reporting and documentation. Also in this issue, we are once again proud to announce the third annual U.S. Government's Top 5 Quality Software Projects contest. You can submit your 2003 nomination at <www.stsc.hill.af.mil> then select the CrossTalk site and click on the Top 5 option.

Special thanks to all of our authors this month for sharing their lessons learned and best practices regarding software quality and defect management. I hope you find this month's issue helpful as you and your teams strive to learn more about handling and managing software defects to meet the end goal of high quality work products for your customers.

Tracy L. Stauder

Tracy L. Stauder
Publisher

ACQUISITION,
TECHNOLOGY
AND LOGISTICS

OFFICE OF THE UNDER SECRETARY OF DEFENSE

3000 DEFENSE PENTAGON
WASHINGTON, DC 20301-3000

14 JUL 2003

MEMORANDUM FOR ALL GOVERNMENT SOFTWARE PROJECTS

SUBJECT: 2003 U.S. Government's Top 5 Quality Software Projects

As the Department of Defense's Executive Agent for Systems Engineering and sponsor for activities aimed at improving software acquisition, I am pleased to announce the search for the 2003 U.S. Government's Top 5 Quality Software Projects.

Many organizations are using processes and practices that result in the successful delivery of projects with significant software content to the United States Government. Looking at past winners of this award, it is apparent that successful projects have used well-defined and proven processes and practices to develop, manage, and integrate software. This award intends to identify successful projects and highlight their efforts.

Access to contest details and to articles discussing previous winners can be found at <www.stsc.hill.af.mil/top5projects>. *CrossTalk* will feature the Top 5 winners in the May 2004 issue, and winners will receive their awards at the 2004 Software Technology Conference. The winning projects will then be highlighted in a series of articles in *CrossTalk's* July 2004 issue.

Mark D. Schaeffer
Director, Systems Engineering
Defense Systems

The Bug Life Cycle

Lisa Anderson
Consultant

Brenda Francis
PowerQuest Corp.

Bugs are everywhere! How do you keep track of them all and still make sure the bugs that need fixing get fixed, the fixed bugs really are fixed, and the little bugs that do not make a difference do not crowd the schedule? Read on to discover how the bug life cycle works and how a database, along with a little organization, will make all the difference in the world.

There are a lot of theories presented at testing seminars. There are a lot of *why test* classes, and a lot of classes on specific techniques, but nothing on a couple of practices that can improve the testing process in a company. We are talking specifically about setting up a defect tracking system and enforcing policies and procedures to resolve those defects. Setting up these two things, more than anything else, can put a company on the road to organizing its testing and quality assurance effort. To fill that gap, we have come up with the *Bug Life Cycle*, as shown in Figure 1.

While we cannot claim it as our own, it is what we have learned over the years as testers. Many of you will find it familiar. Anyone who can figure out that the software is not working properly can report a bug. The more people who critique a product, the better it will be. However, here is a short list of people who are expected to report bugs:

- Testers/Quality Assurance Personnel.
- Developers.
- Technical Support.
- Beta Sites.
- End Users.
- Sales and Marketing Staff (especially when interacting with customers).

When do you report a bug? When you find it! Waiting means that you might forget to write it down altogether, or important details about the bug can be forgotten. Writing it now also gives you a *scratch pad* to make notes on as you do more investigation and work on the bug.

Writing the bug when you find it makes that information instantly available to everyone. You do not have to run around the building telling everyone about the bug; it is in the database. Additionally, the information about the bug does not change or get forgotten with every telling of the story.

The easiest way to keep track of defect reports is in a database. Keeping track on paper works, but paper can get lost or destroyed; a database is more reliable and can be backed up on a regular basis.

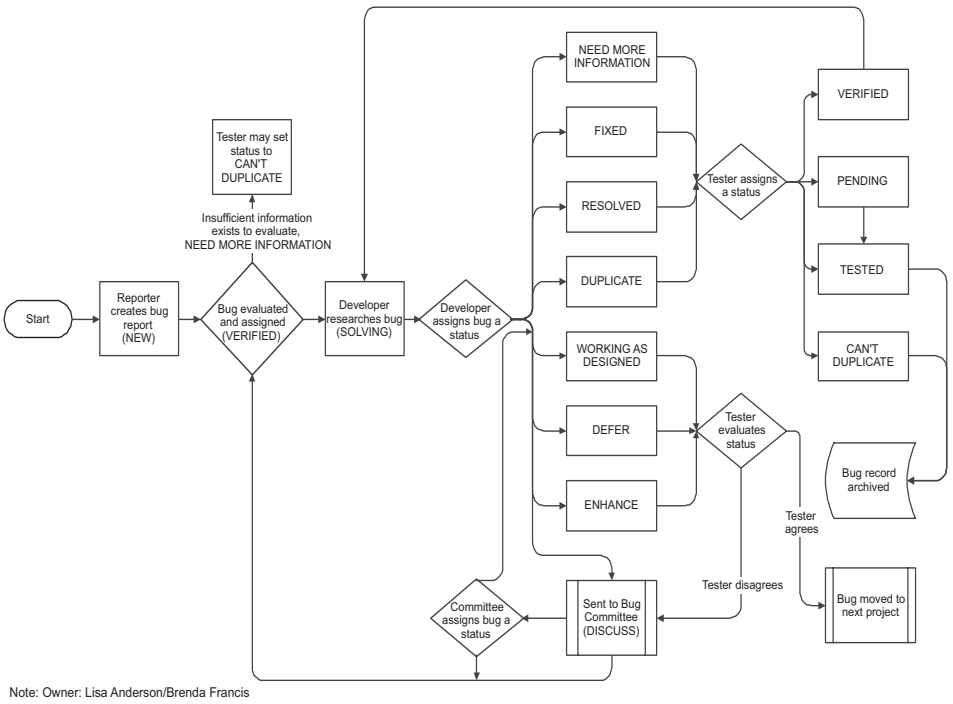
You can purchase many commercially available defect-tracking databases, or you can build your own. It is up to you. We have always built our own with something small like Microsoft Access or SQL Server. It was cheaper to build and maintain it on site than to purchase it. You will have to run the numbers for your situation when you make that decision.

The rule of thumb is one (and only one) defect per report (or record) when writing a bug report. If more than one defect is put into a report, the tendency is to deal with the first problem and forget the rest. Remember that defects are not always fixed at the same time. With one defect per report, as the defects get fixed, they will be tested individually instead of in a group where the chance that a defect is overlooked or forgotten is greater.

A good bug report includes the following items:

- Put the *reporter's name* on the bug. If there are questions, you need to know who originated the report.
- Specify the *build* or *version number* of the code being worked on. Is this the shipping version or a build done in-house for testing and development? Some bugs may only occur in the shipping version; if this is the case, the version number is a crucial piece of information.
- Specify the *feature* or *specification* or *part of the code*. This facilitates getting the bug to the right developer.
- Include a *brief description* of what the problem is. For example, *fatal error when printing landscape* is a good description; it is short and to the point.
- List *details*, including how to duplicate the bug and any other relevant data or clues about the bug. Start with how the computer and software are set up. List each and every step (do not leave anything out). Sometimes a minor detail can make all the difference in duplicating or not duplicating a bug. For example, using the keyboard versus using the mouse may produce very different results when duplicating a bug.
- If the status is not *new* by default, change it to *new*. This is a flag to the bug verifier that a new bug has been created and needs to be verified and assigned.

Figure 1: *The Bug Life Cycle*



Rating	Value
Blue Screen/Hang	1
Loss Without a Workaround	2
Loss With a Workaround	3
Inconvenient	4
Enhancement	5

Table 1: *Bug Severity*

Rating	Value
Always	1
Usually	2
Sometimes	3
Rarely	4
Never	5

Table 2: *Bug Likelihood*

Things to Remember

Keep the text of the bug impersonal. Bug reports will be read by a variety of people, including those outside the department and possibly the company. Please do not insult people's ancestors, their employer, the state where they live, or make any other impulsive or insensitive comment. Be careful with humorous remarks; one person's humor is another person's insult. Keep the writing professional.

Be as specific as possible in describing the current state of the bug along with the steps to get into that state. Do not make assumptions that the reader of the bug will be in the same frame of mind as you are. Please do not make people guess where you are or how you got into that situation. Not everyone is thinking along the same lines as you are.

Prioritizing Bugs

While it is important to know how many bugs are in a product, it is even more useful to know how many are severe, ship-stopping bugs compared to the number of inconvenient bugs. To aid in assessing the state of the product and to rank bug fixes, bugs are prioritized. The easiest way to prioritize bugs is to assign each bug a severity rating and a likelihood rating. The bug reporter does this assignment when the bug is created. Each severity and likelihood category has an associated value. The bug's priority is calculated by multiplying the value of the severity and likelihood ratings.

The severity tells the reader how bad the problem is. Or in other words, it tells what the results of the bug are. Table 1 shows a common list for judging the severity of bugs. Sometimes there is disagreement about how bad a bug is.

To determine how likely it is for a bug to occur, put yourself in the *average* user's place. While the tester may encounter this bug every day with every build, if the user is not likely to see it, how bad can the bug be? Table 2 shows a rating of bug likelihood.

Severity x Likelihood = Priority

To compute the priority of a bug, multiply the numeric value given to the severity and the likelihood. Do the math by hand or let your defect tracker do it for you. The trick is to remember that the lower the number, the more severe the bug is. The highest rating is a 25 (5 x 5), the lowest is 1 (1 x 1). The bug with a 1 rating should be fixed first, while the bug with a 25 rating may never get fixed.

This system is just the beginning. A more sophisticated or advanced way of prioritizing bugs would be to weigh the features and add a development risk value. Each feature adds a different value to the product. Some features are more important than others are. In order to weigh the features, consider each feature's contribution to the product, and weigh it accordingly on a scale of one to five.

Development risk encompasses a number of things. How risky is it to fix a specific piece of code? How will this fix affect the rest of the code base? If it is a minor fix but affects most of the files in the code base by forcing a recompile, then it is a serious fix. This kind of fix could force regression testing that could add significant time to the schedule. Many features may depend on this base feature; this would increase the development risk. If the fix is to a help file that does not affect any other files, then it is a minor one and may be of acceptable risk. This seems like a lot of questions, but the answers can help you assign the proper development risk to each feature.

Using these algorithms may cause bug priorities to cluster around certain values. If you notice this is occurring, you can adjust the algorithm accordingly using a *fudge factor* but that is beyond the scope of this article.

A listing of these bugs ordered by rating means the most important ones will be at the top of the list and should be dealt with first. Sorting bugs this way lets management know whether the product is ready to ship or not. Use whatever criteria you select such as, *all bugs with a priority of 10 or less must be fixed*. If the number of these bugs is zero, the product can ship. If there are any severe bugs, then bug fixing must continue.

Other Useful Information

- Who is the bug *assigned to*? Who is going to be responsible for fixing the bug?
- What *platform* was the bug found on (B, Windows, Linux, etc.)? Is the bug specific to one platform or does it occur on all platforms?
- What *product* was the bug found in? This is important if your company is doing multiple products.
- What *company* would be concerned about this bug? If your company is working with multiple companies, this is a good way to track that information.
- Whatever else you want or need to keep track of. Some of these fields can also have value to marketing and sales. It is a useful way to track information about companies and clients.

Now We Have a Bug

At this point, it may be helpful to have access to the bug life-cycle chart and refer to it during the following discussion. Some paths that a bug may take can be confusing; the chart helps simplify the process.

The first step after the bug is created is *verification*. A bug verifier searches the database for all bugs with a *New* status. He duplicates the bug by following the steps listed in the *details* section of the bug. If the bug is reproduced and has all the proper information, the *Assigned To* field is changed to the developer who will be fixing the bug, and the status is changed to *Verified*. If the bug is not written clearly, is missing steps, or cannot be reproduced, it will be sent back to the bug reporter for additional work.

The Assigned To field contains the name of the person responsible for that area of the code. It is important to note that from this point forward, the developer's name stays on the bug. Why? There are usually more developers than there are testers. Developers look at bugs from a standpoint of *what is assigned to me?*. Testers have multiple features to test, which means that testers look at bugs from a standpoint of *what needs to be tested?* Because of the different way testers and developers work, developers sort bugs by the Assigned To field and testers sort bugs by the Status field. Leaving the developer's name on the bug also makes

it easier to send the bug back to the developer for more work. The tester simply changes the Status field to Verified, and then automatically goes back to the developer.

The first thing the developer does is give the bug a *Solving* status indicating that he has seen the bug and is aware that it is his responsibility to resolve it. The developer works on the bug and, based on his conclusions, assigns a status to the bug indicating what the next step should be.

Remember, the developer does *not* change the Assigned To field. His name stays on the bug in case the bug has to go back to him; it will make it back to his list. This procedure ensures that bugs do not fall between the cracks. The following paragraphs list statuses that a developer can assign to a bug.

The *Fixed* status indicates that a change was made to the code and will be available in the next build. Testers search the database on a daily basis looking for all Fixed-status bugs. Then the bug reporter or tester assigned to the feature retests the bug, duplicating the original circumstances. If the bug passes, it gets a *Tested* status. If the bug does not pass the test, it is given a Verified status and sent back to the developer with information about the test performed (for example, the build that was used to test the fix). Notice here that since the bug's Assigned To field has retained the developer's name, it is an easy process for the tester to send the bug back by simply changing the status to Verified.

The *Duplicate* status bug is the same as a previously reported bug. Sometimes only the developer or someone looking at the code can tell that the bug is a duplicate; it is not always obvious from the surface. A note referencing the previous bug number is placed on the duplicate bug. A note is also placed on the original bug indicating that a duplicate bug exists. When the original bug is fixed and tested, the duplicate bug will be tested also. If the bug really is a duplicate, when the original bug is fixed the duplicate bug will be fixed as well. If this is the case, both bugs get a Tested status.

If the duplicate is still a bug – while the original bug is working properly – the duplicate bug does not keep its Duplicate status. It gets a Verified status and is sent back to the developer. This is a *fail-safe* built into the bug life cycle. It is a check and balance that prevents legitimate bugs from being swept under the carpet. However, here is a note of warning: Writing lots of duplicate bugs can give a tester a bad reputation. It pays to set time aside daily to read all the new bugs written the previous day to avoid re-reporting bugs.

Resolved means that the problem has been resolved but no code has changed. For example, bugs can be resolved by getting new device drivers or third-party software. Resolved bugs are tested to make sure that the problem really has been resolved with the new situation. If the problem no longer occurs, the bug gets a Tested status. If the Resolved bug still occurs, it is sent back to the developer with a Verified status.

Need More Information indicates that the bug verifier or developer does not have enough information to duplicate or fix the bug; for example, the steps to duplicate the bug may be unclear or incomplete. The developer changes the status to Need More Information and includes a question or comment to the reporter of the bug. This status is a flag to the bug reporter to supply the necessary information or a demonstration of the problem. After updating the bug information in the *Notes* field, the status is put back to Verified so the developer can continue working on the bug. If the bug reporter can no longer duplicate the bug, it is given a *Can't Duplicate* status along with a note indicating the circumstances.

It is important to note that the only person who can put Can't Duplicate on a bug is the person who reported it (or the person testing it). The developer *cannot* use this status; he must put Need More Information on it to give the bug reporter a chance to work on the bug.

This is another example of a *fail-safe* built into the database. It is vital at this stage that the bug be given a second chance. The developer should never give a bug a Can't Duplicate status. The bug reporter needs an opportunity to clarify or add information to the bug or to retire it.

The developer may want to protest the bug: Should it be included in this version of the product, or perhaps not be fixed at all? The status *Discuss* is used to send the bug to the Bug Committee (test manager, development lead, and/or development manager) for further discussion. The developer should be sure to include comments about why the bug is being protested or needs further discussion.

If the developer has examined the bug, the product requirements, and the design documents, and determined that the bug is not a bug, it is *Working as Designed*. In other words, what the product or code is doing is intentional as per the design. Or as someone more aptly pointed out it is *working as coded!*

This bug can go several directions after being assigned this status. If the tester agrees, the status remains and the bug is finished. The bug may be sent to documentation for inclusion in the help files and manual. If the tester disagrees, the bug can be appealed by putting a Discuss status on it to send the bug to the Bug Committee. The tester should include in the notes a reason why, although the developer has given it a *Terminal* status, it should be changed now. The bug may also be sent back to the design committee so that the design can be improved.

Working as Designed is a dangerous status. It is an easy way to hide annoying bugs. It is up to the bug reporter to make sure the bug does not get forgotten. Product managers may also review lists of bugs recently assigned Working as Designed.

The *Enhance* status means that while the suggested change is a great idea, because of technical reasons, time constraints, or other factors, it will not be considered until the next version of the product. This status can be appealed by changing the status to Discuss and adding a note specifying why it should be fixed now.

Defer is almost the same status as Enhancement. This status implies that the cost of fixing the bug is too great given the benefits it could produce. If the fix is a one-liner to one file that does not influence other files, it might be okay to fix the bug. On the other hand, if the fix will cause the rebuild of many files that would force product retesting and there is no time to test the fix before shipping the product, then the fix would be unacceptable and the bug would get a Defer status. To appeal the status, send it back through the process again by putting a Discuss status on it with a note stating why it should be fixed now.

You may see the *Not to be Fixed* status although we do not recommend making this status available for use. There may be extenuating circumstances where a bug will not be fixed because of technology, time constraints, a risk of destabilizing the code, or other factors. A better status to use is Enhance. To appeal the status, send it back through the process again by putting a Discuss status on it with a note saying why it should be fixed now.

This is similar to the Working as Designed status in that its use can be dangerous. Be on the watch for this one. Sometimes developers call this status the *you can't make me* status.

The Tested status is used only by testers on Fixed, Resolved, and Duplicate bugs. This status is an *end-of-the-road* status indicating that the bug has been verified as Fixed; the bug has now reached the end of its life cycle.

The *Pending* status is used only by testers on Fixed bugs when a bug cannot be tested immediately. The tester may be waiting on hardware, device drivers, a build, or addi-

tional information necessary to test the bug. When the necessary items have been obtained, the bug status is changed back to Fixed and is tested. It is critical that the bug is tested just as thoroughly as any other bug fix; make sure testing is not skipped.

The Can't Duplicate status is used only by the bug reporter; developers and managers cannot use this status. If a bug is not reproducible by the assigned developer or bug verifier, the bug reporter needs a chance to clarify or add to the bug. There may be a hardware setup or situation, or a particular way of producing a bug that is peculiar to a specific computer or bug reporter and he needs a chance to explain what the circumstances are. Limiting the use of this status to bug reporters prevents bugs from slipping between the cracks and not getting fixed.

It is important to note that before shipping a product, all active bugs must be addressed; that is, all bugs with a Fixed, Need More Information, Resolved, or Pending status must be taken care of. You should also set a criteria based on bug priority; for example, the number of active bugs rated five or less must be zero. These criteria are excellent benchmarks for judging the readiness of a product.

What Happens After Shipping?

All bugs with a Tested or Can't Duplicate status are archived. This means that the records are either removed and placed in an archive database, or are flagged to be hidden from the current database view. Never delete any bug records; it may be necessary to do some historical research in the bug file (*What did we ship when?* or *Why did we ship with this bug?*).

Bugs with Enhance and Defer status are moved to the New bug file or retained in the current bug file. The statuses of these bugs are then changed back to Verified.

This methodology not only shortens the list of bugs to deal with, but it also moves bugs that were not considered necessary for the current product to ship into consideration for the next version of the product.

Reports

The data in the bug file are not very useful until sorted and presented in an organized fashion; they then become information. For example, sorting by developer, the information becomes a *to-do* list sorted by rating. Sorting by status lets the reader know how many bugs are submitted or in progress; sorting by feature asks, "How many open bugs are there for a particular feature?" "What feature needs more work?" and "What feature is stable?" Sorting by product is useful when more than one product is being worked on simultaneously.

Be aware that there are certain metrics or reports that should not be used. If you use these reports you will destroy the credibility of your bug file and it will be reduced to a *laundry list* for developers. One of these reports is "*How many bugs did a tester report?*" and the other is "*How many bugs did a developer fix*?" Neither one of these has any useful purpose except to beat up people uselessly [1].

A defect database that has all these fields built into it and has a good query language is able to sort defect data and turn it into useful information. Setting up customized queries should not be too difficult for the average database administrator.

Conclusion

The challenges of following a bug life cycle are far outweighed by the benefits derived. A well planned and closely managed defect database not only tracks current defects against any number of builds and/or products, it also provides a virtual paper trail for the overall progress of a product as it is coded, tested, and released. If sufficient time is provided for building a defect tracker that works for your company, it is more likely you will release a less buggy product, or at least a product where most of the big ones have *not* gotten away.◆

Reference

1. Kaner, Cem, et. al. Testing Computer Software. 2nd ed. New York: International Thomson Computer Press, 1993.

Additional Reading

1. Beizer, Boris. Software Testing Techniques. 2nd ed. New York: International Thomson Computer Press, 1990.
2. Hetzel, Bill. The Complete Guide to Software Testing. 2nd ed. New York: John Wiley & Sons, Inc., 1988.
3. Jones, Capers. Software Quality: Analysis and Guidelines for Success. New York: International Thomson Computer Press, 1997.
4. Kit, Edward. Software Testing in the Real World. New York: Addison-Wesley, 1995.
5. Mirrer, Barry. "Organize Your Problem Tracking System: Cleaning Up Your Bug Database Can Be as Easy as Organizing Your Sock Drawer." Software Testing Quality Engineering Sept./Oct. 2000: 34-39.
6. Myers, Glenford J. The Art of Software Testing. New York: John Wiley & Sons, Inc., 1979.
7. Patton, Ron. Software Testing. Indianapolis: Sams, 2000.
8. Institute of Electrical and Electronics Engineers, Inc. IEEE Standard for Software Test Documentation 829-1998. New York: Institute of Electrical and Electronics Engineers, Inc., 1998.

About the Authors

Lisa Anderson has been a tester and quality assurance engineer with WordPerfect Corp., Novell, Inc., Corel, Inc., and PowerQuest Corp. Anderson has also been a director of Quality Assurance (QA) for a small startup company and was a QA manager at PowerQuest Corp. She has been in the software QA field since 1991; has attended STAR East 1999, 2000, and 2001; participated in Software Testing Manager Roundtable 4 and 5; and is the sponsor of Mountainwest Enterprise Testing Roundtable 2003. Anderson has bachelor's degrees in education and computer information systems and is currently working on a master's degree in computer information systems.

Phone: (801) 319-5840
E-mail: lisaan@lisaan.com

Brenda Francis is a software quality engineer at PowerQuest Corp. She worked formerly for Novell and WordPerfect in problem resolution teams and has been in the software quality assurance field since 1997. She has a bachelor's degree in international relations and a master's degree in American history.

PowerQuest Corp.
P.O. Box 1911
Orem, UT 84059-1911
Phone: (801) 437-8900
E-mail: brenda.francis@powerquest.com

Comparing Lean Six Sigma to the Capability Maturity Model

Dr. Kenneth D. Shere
The Aerospace Corporation

The Capability Maturity Model® has been widely used by the government to evaluate contractors as part of the acquisition process for large, complex systems and has been used by contractors to improve their software processes. Whereas this approach makes sense, both the government and industry can do better by including Lean Six Sigma in their process improvement and acquisition approaches. In this article, the two concepts are compared; examples of organizations using Lean Six Sigma for software are presented.

The Software Engineering Institute (SEI) initially developed the Capability Maturity Model® for Software (SW-CMM®) [1] with the initial purpose of providing a map for improving software processes. The SW-CMM also provides a basis for assessing the maturity of an organization's software processes. Because of its success, other capability maturity models were developed. These include the following:

- A Software Acquisition CMM by the SEI [2].
- A testing capability maturity model [3].
- Several systems engineering capability maturity models [4, 5, 6].

Due to the growing variety of capability maturity models, the SEI developed a consolidated approach called the CMM Integration℠ (CMMI®) [7]. Capability maturity models have been a topic of many articles in CrossTalk. In this article, capability maturity model is used generically. When a specific capability maturity model is intended, it is identified explicitly.

Lean Six Sigma™ (LSS) is a systems-engineering approach to defining, measuring, analyzing, and improving processes. LSS was initially developed for manufacturing, but has been successfully applied to all types of processes – including transactional processes, services, and software. A brief introduction to this topic is given in [8, 9].

It is assumed here that the reader has a reasonable familiarity with capability maturity models and has at least an introductory knowledge of LSS. The purpose of this article is to compare capability maturity models and LSS[1].

In the first section, key features of these two are contrasted. Having looked at their differences, the next section will focus on success factors. Lastly, two examples are presented in which industry has used Six Sigma in conjunction with capability maturity models. This article ends with some conclusions and recommendations.

® Capability Maturity Model, CMM, and CMMI are registered in the U.S. Patent and Trademark Office.

℠ CMM Integration is a service mark of Carnegie Mellon University.

™ Six Sigma is a trademark of Motorola, Inc.

Contrasting Capability Maturity Models and LSS

The following sections compare various attributes of capability maturity models and LSS. These attributes include institutionalization, assessment approaches, focus, and measurement. The primary differences between capability maturity models and LSS derive from the fact that capability maturity models are *models*, whereas, LSS is a *method*.

> *"LSS [Lean Six Sigma] was initially developed for manufacturing, but has been successfully applied to all types of processes – including transactional processes, services, and software."*

Basis

Capability maturity models are models; they focus on *what*. The SEI's CMM specifies that policies, procedures, and guidelines be explicitly defined, including Key Process Areas (KPAs), goals for each KPA, and practices associated with each KPA. The CMM defines maturity in terms of whether or not management and engineering processes have been defined, implemented, and consistently used throughout the organization. The CMM has an underlying assumption that defined processes are good. It does not provide a procedure for defining or evaluating processes. Statistical methods are not explicitly specified by the CMM. Experience has shown that the CMM influences management's behavior, but engineers seem to perform the same way regardless of the capability level of the organizations [10].

LSS is a methodology; its focus is on *how*. In a sense, LSS is simply codified good systems engineering. One of the foundations of LSS is statistical quality control; LSS defines process performance in terms of its mean and variance. A concept that permeates the method is reducing the cost of poor quality. This concept is viewed at the broadest possible level. LSS does not explicitly provide a list of procedures and policies needed by an organization.

Institutionalization

Both the SEI's CMM and LSS recognize that institutionalizing processes is a key to success, but their approaches are different. The CMM requires institutionalization by specifying the following:

- Written organizational policies that exist regarding the use of engineering and management processes.
- Adequate resources are provided for implementing processes.
- Appropriate oversight is provided (which could have the form of either taking certain measurements or management reviews).

LSS does not ask the question of whether a process is institutionalized. It is successful only when LSS itself is institutionalized. Specifically, LSS requires an extensive training program. All lead managers and engineers are expected to become experts in LSS, and are frequently referred to as Six Sigma *black belts*. This status requires taking a four- to six-week course over four months while applying what is learned in the course to a specific process improvement task. Following this training, the trainee is required to lead two more tasks and then take a test to be

certified.

With LSS, everyone in the organization is trained to a level that is job-dependent. Training could be a one-week course with application to a specific task for Six Sigma *green belt* status. Other training is at the executive level in which people take one- to three-day courses to obtain a basic understanding of the process.

Institutionalization is obtained by training and application throughout the organization. Institutionalization is impossible to obtain for either LSS or the CMM unless there is a long-term, substantial corporate commitment.

Assessment Process Control

The CMM has the advantage of being controlled by the SEI, which has developed a substantial body of material for use in conducting capability maturity assessments, and has conducted many of these assessments. The SEI provides training courses in this area, and people can be certified as software capability evaluators. Generally, an external auditor assesses the *CMM level* by inspecting several projects across an organization. The organization provides all requested documentation for review and access to key people for interviews.

In the case of government procurements, the CMM assessments provide an indication of an organization's maturity based on other projects, but do not guarantee that the same processes and approach will be used for the system being procured. To remedy this problem, some acquisitions require a periodic CMM assessment of the contractor's effort during system design and development. The results of these assessments are (theoretically) tied to award fees.

LSS has no organization that is considered either the governing body or the standard bearer. Consequently, every Six Sigma organization defines it somewhat differently. This situation exists because the development and use of LSS has been driven by industry – in contrast to the CMM whose development was funded by the government and implemented by a federally funded research and development center. No external body exists to declare whether or not an organization is LSS. Nonetheless, there are recognized best practices associated with Six Sigma.

To complicate matters, Six Sigma organizations do their own certification. Thus, certification from one organization might not be accepted by another organization. In practice, anybody who is certified by one company is generally recognized as a Six Sigma expert by other organizations. However, if a certified expert (a Six Sigma black belt) changes organizations, he or she still needs to take Six Sigma training at the new organization to assure that he or she would be applying the methodology consistently with other people in the new organization.

If an organization claims to be a LSS organization in a proposal, assessing the veracity of this claim is relatively straightforward. The buyer could conduct a review of (1) the organization's training and certification program, (2) the certification of people committed to the program, and (3) the process documentation and performance data (for all processes to be used in the proposed acquisition).

> *"The primary criterion used in assessing whether a process is lean is to determine whether each activity in the process adds value – i.e., it provides something the customer is willing to pay for."*

Focus

The CMM is introspective. This focus is due to the nature of the model. Assessments determine whether measurements are being taken, policy exists, resources are applied, people are trained in the process, and products are reviewed internally. When the model looks outward, as it does in the Subcontract Management KPA, it is from the perspective of whether the internal management processes and policies exist to handle subcontracts.

LSS is inherently focused outward. The primary criterion used in assessing whether a process is *lean* is to determine whether each activity in the process adds value – i.e., it provides something the customer is willing to pay for. The *Six Sigma* part of LSS looks at the *cost of poor quality*. This criterion is directly tied to customer satisfaction and the supply chain (including subcontractors). Many companies also tie this criterion to their business plans and strategic goals.

Measurement

The process improvement approach of Six Sigma is partitioned into five phases: define, measure, analyze, improve, and control. Having defined an existing process in the first phase, the next phase is to measure its performance. Performance measurements of throughput and quality are taken. Throughput is the number of items produced, services rendered, etc. Wait time and cues are also measured. Quality is expressed statistically as the process mean and variation. The cost of each step of the process is measured in terms of currency, time, and resources. The physical layout between process *stages* is measured to determine wait time and cost, or transportation expense between stages.

Various analyses are then performed, which include defect analysis (for example, cause and effect or fishbone charts) and analysis of variance. Simulations based on experiments' design are performed to determine candidate improvements. During the improvement phase, a prototype or initial improvement is made and measured. The results are compared with the simulation results to validate the improvement before it is implemented for the process. The improvement is implemented as an operational change in a controlled manner while measurements are taken to validate the prototype results. Measurement is a way of life in Six Sigma.

For measurement in capability maturity models, this discussion shall focus on the CMMI because it is the most recent and comprehensive model [11]. Measurement permeates throughout the CMMI. In the *staged model*, Level 4 is *Quantitatively Managed*. The purpose of this level is to obtain the data needed for the organization to effectively optimize its processes. Level 5 is *Optimization*. It is clear from thinking about the purposes of Levels 4 and 5 that at their core the CMM and Six Sigma have a great deal in common.

Unlike other capability maturity models, the CMMI has a *process area* (PA) *Measurement and Analysis*. This PA [12] specifies that a measurement capability be established to support management needs. The Measurement and

Analysis PA is oriented toward systematically collecting typical program data (defect density, activity logs, peer review coverage, and so on). Measuring process capability, as such, could be included in this PA, but it is not a core purpose.

In the CMM, common features that contain key practices organize each KPA. The common features are ability to perform, activities performed, measurement and analysis, and verifying implementation. The CMMI slightly modifies the common features by replacing *directing implementation* with *measurement and analysis*. The CMM documentation is good at indicating the types of items that might be measured for each process, but does not explicitly say what to measure. The CMM documentation indicates that analysis of the data is necessary, however, neither type of analysis nor analytical procedures are explicitly discussed.

Success Factors for Lean Six Sigma and the CMM

Both LSS and CMM are based on institutionalizing defined processes, performing quantitative measurement of the processes, and improving the processes based on these measurements. Both approaches address the systemic problems that have existed in our approach to software and systems engineering. Neither approach will be successful unless a substantial corporate commitment is made. This commitment includes the following:

- No-nonsense leadership from the top.
- Training (to various levels) of *everybody* in the organization.
- An up-front financial investment to get the process started.
- Organizational recognition of the importance of a capability maturity model or LSS.
- Rewarding people who are successfully implementing capability maturity models or LSS.

Organizational recognition does not mean that a big bureaucracy is needed. For example, Dow Chemicals had 2001 sales of $27.8 billion; they have more than 50,000 employees distributed over more than 40 countries. Six Sigma is implemented throughout the company with training materials in 13 languages. More than 90 percent of Dow employees will be involved with Six Sigma in some way by 2003 [13]. Their corporate staff for Six Sigma is about five people. There are also a few *staff-level* people in each of their operating businesses.

Rewards are critical because employees pay attention to a leader's actions more than his or her words. When rewards are primarily given to people for being a hero – working a large number of problems to save a program in trouble – that is what people believe is expected. Rewards need to be given primarily to people who did the job right in the first place, i.e., within budget and schedule.

Both approaches have been used successfully. The SW-CMM Level 5 organizations have the data to prove that they can deliver projects on time and within budget. It has been reported that variation between the actual cost and schedule to the estimated cost and schedule for projects performed by these organizations is usually within 3 percent [14]. Even Level 3 organizations have benefited dramatically from SW-CMM. For example, John Vu of The Boeing Company has provided statistics that demonstrated variation of labor hours went from historical figures (Levels 1 and 2) of *+20 percent to –145 percent* to a Level 3 variance of *+20 percent to –20 percent* [15]. He also provided data to show that simply implementing a formal review and inspection procedure caused an increase of design effort by four percent and a decrease of rework by 31 percent. That change represents a cost benefit ratio of 1:7.75 – almost an order of magnitude.

"Rewards need to be given primarily to people who did the job right in the first place, i.e., within budget and schedule."

Corporate presidents have discussed the benefits of LSS in terms of profit added to the bottom line. For example, at the 1999 Annual Meeting of General Electric, Jack Welch said that the Six Sigma effort at GE had already saved $3.5 billion beyond their investment of $1 billion, and they were just at the knee of the curve [16].

Integrating Lean Six Sigma and the CMM

These two approaches to process improvement have the same goal. In fact, if an organization is truly a CMM Level 5 organization, it is also in spirit, if not in fact, a Six Sigma organization. Conversely, a true Six Sigma organization is in spirit, if not in fact, a CMM Level 5 organization. In each case, processes must be defined, data must be collected, and data used quantitatively to improve the processes. Some organizations do not begin integrating LSS with CMM until Level 3 has been attained (so processes have been defined), whereas others use LSS techniques to help define processes during the lower levels of maturity.

Examples of companies that have integrated Six Sigma with the CMM are Motorola, Tata Consultancy Services (TCS), Honeywell, and PS&J Software Six Sigma.

Motorola Labs used multivariate analysis techniques of Six Sigma to determine the causes of delays in closure of corrective action reports, and to improve their audit process. How to apply multivariate techniques to software processes is included in the Motorola University I-Cubed Presentation Series [17]. Motorola has several facilities evaluated at CMM Level 5, and is the founder of Six Sigma.

TCS also combined Six Sigma with the CMM. They specifically applied Six Sigma to their software review process and to decisions on program metrics [18]. This work was done for their Chennai, India, engineering center for General Electric. This TCS center has been evaluated as a CMM Level 5 organization.

Honeywell and PS&J Software Six Sigma introduced Six Sigma techniques into the Personal Software Process as defined by Watts Humphrey at the Software Engineering Institute [19].

Conclusions

The SEI's CMM and LSS have independently changed the way many major corporations think about their processes by addressing systemic problems in a constructive manner. These approaches are complementary. They both apply to the acquisition and development of complex systems. Their successful application depends on committed leaders, training, institutionalization, demonstrating a positive return on investment, and continuous reinforcement and reward.◆

References

1. Humphrey, Watts. Managing the Software Process. Reading, MA:

COMING EVENTS

October 15-18
Richard Tapia Diversity in Computing Conference
Atlanta, GA
www.ncsa.uiuc.edu/Conferences/Tapia2003

October 16-17
Six Sigma Software Development Conference
Boston, MA
www.frallc.com/infotech.asp#c112

October 17-19
Pop! Technology Conference
Camden, MA
www.poptech.org

October 20-24
Quality Assurance Joint Conference on Compressing Software Development Time
Baltimore, MD
www.qaiusa.com

October 21-23
7th IEEE International Symposium on Wearable Computers
White Plains, NY
www.cc.gatech.edu/ccg/iswc03

October 27-31
STARWEST '03
San Jose, CA
www.sqe.com/starwest/

November 2-5
Amplifying Your Effectiveness Conference
Phoenix, AZ
www.ayeconference.com

November 17-21
2nd International Conference on Software Process Improvement
Washington, DC
www.icspi.com

April 19-22, 2004
Software Technology Conference 2004

Salt Lake City, UT
www.stc-online.org

Addison-Wesley, 1990.
2. Cooper, Jack, and Matthew Fisher, Eds. Software Acquisition Capability Maturity Model® (SA-CMM®). CMU/SEI-2002-TR-010. Pittsburgh, PA: Software Engineering Institute, Mar. 2002.
3. Burnstein, Ilene, Taratip Suwannasart, and C. R. Carlson. "Developing a Testing Maturity Model." CrossTalk 9.8, 9.9 (Aug., Sept. 1996).
4. Shere, Kenneth D., and Mark J. Versel. Extension of the SEI Software Capability Model to Systems. Proc. of the 18th Annual International Computer Software and Applications Conference, Los Alamitos, CA, 1994. New York: IEEE Computer Society Press, 1994: 195-200.
5. Software Productivity Consortium. A Systems Engineering Capability Maturity Model Ver. 1.0. SPC-95007-CMC. Herndon, VA: Software Productivity Consortium, May 1995.
6. Arunski, Karl, et. al. Systems Engineering Capability Model. EIA/IS 731. Arlington, VA: Electronic Industries Alliance, 17 Jan. 1999 <www.geia.org/sstc/G47/731 dwnld.htm>.
7. Software Engineering Institute. Capability Maturity Model Integration (CMMI®), Ver. 1.1: CMMI for Systems Engineering, Software Engineering, Integrated Product and Process Development, and Supplier Sourcing V1.1 CMMI-SE/SW/IPPD/SS. Pittsburgh, PA: Software Engineering Institute, Mar. 2002.
8. Shere, Kenneth D. "Lean Six Sigma – How Does It Affect the Government?" CrossTalk 16.3 (Mar. 2003): 8-11.
9. Siviy, Jeannine. Six Sigma: Software Technology Review. Pittsburgh, PA: Software Engineering Institute, 1 May 2001 <www.sei.cmu.edu/str/descriptions/sigma6_body.html>.
10. Humphrey, Watts S. "What Is Excellence?" International Conference on Software Process Improvement. College Park, MD, Nov. 2002 <www.software-process-institute.com>.
11. Software Engineering Institute. Capability Maturity Model Integration (CMMI®), Version 1:1: CMMI for Systems Engineering, Software Engineering, Integrated Product and Process Development, and Supplier Sourcing V1.1 CMMI-SE/SW/IPPD/SS. Pittsburgh, PA: Software Engineering Institute, Mar. 2002.
12. Ibid: 163-180.
13. Parker, Mike. "Special Commemorative Issue." Around Dow. Midland, MI: Dow Chemical Company, 2000: 31, 44 <www.dow.com/webapps/lit/litorder.asp?objid=09002f13800ba251&filepath=/noreg>.
14. Johnston, Margaret. "Integrators Aim High on Software Methodology." Federal Computer Week 25 Jan. 1999 <www.fcw.com/fcw/articles/1999/FCW_012599_53.asp>.
15. Vu, John. "What Justifies a Rating of CMM Level 5?" Software Engineering Process Group Conference. New Orleans, LA, Mar. 2001.
16. Welch, Jack. "Presentation of the Chairman of the Board." Annual Meeting. Fairfield, CT: General Electric, 1999.
17. McCarty, Tom. Private Communication. Motorola, 2003.
18. Moorthy, Vinay. Private Communication. Tata Consultancy Services, 2003.
19. George, Ellen, and Steve Janiszewski. "SPC Is the Perfect Tool for PSP Post-Mortem Data Analysis." PS&J Software Six Sigma, 2001 <www.SoftwareSixSigma.com>.

Note

1. For another comparison of Six Sigma to the Capability Maturity Model, cf. Card, David. "Sorting Out Six Sigma and the CMM." IEEE Software. May/June 2000: 11-13.

About the Author

Kenneth D. Shere, Ph.D., is a senior engineering specialist at The Aerospace Corporation where he provides systems and software engineering, acquisition, and strategic leadership support to various government organizations. He is certified as a Lean Six Sigma green belt and a Software Engineering Institute Software Capability Evaluator. Shere has published 18 articles and two books. He has a bachelor's of science degree in aeronautical and astronautical engineering, a master's of science degree in mathematics, and a doctorate in applied mathematics, all from the University of Illinois.

The Aerospace Corporation
15049 Conference Center Drive
Chantilly, VA 20151
Phone: (703) 633-5331
Fax: (703) 633-5006
E-mail: kenneth.d.shere@aero.org

Managing Software Defects in an Object-Oriented Environment

Houman Younessi
Rensselaer Polytechnic Institute-Hartford Graduate Campus

Managing defects when developing object-oriented systems has its own challenges. In this article, the impact of adopting the object-oriented paradigm on how we manage defects at various phases of the software process is discussed. Specific issues relating to both the structural implications and the environmental and process considerations are named and discussed, with solutions provided.

The benefits of the object-oriented (OO) paradigm are well publicized. Less so are the areas where this paradigm actually creates challenges and difficulties. One such area is defect management. This article looks at why object-orientation might present such challenges, and what it is like to manage defects in an OO environment.

In general, OO systems score lower in terms of testability compared to procedural systems [1]. The reasons for such low testability can be traced to the structural composition of OO systems discussed in the following sections. Each section begins a brief definition of the issue under discussion, a short synopsis or some explanation closely relating to the issue at hand, and a description of the defect management problems that are relevant. Each problem is numbered along with its corresponding solution(s), for example, (1) Abstraction Reduces Observability. Keep in mind there may be more than one corresponding solution to each preceding problem.

Abstraction

Abstraction is that essential property that allows the selection of a logical and coherent conceptual boundary so that the object is identifiable by its essential characteristics; these are those characteristics that define what it is and what it does without heed to how such is accomplished. While abstraction is of potentially great benefit to the modeler, the designer, the user, the maintainer, and the reuser, it is often of hindrance to the defect manager.

Problems

(1) Abstraction Reduces Observability [1]. Observability, or internal state visibility, is the ability to examine the internal state of an object at any one time. Given that abstraction, in essence, masks access to a lot of this information, it dramatically reduces observability.

(2) Partial/Distributed Implementation. An abstract class is one in which the implementation of at least one feature is deferred to another class. This creates a problem in testing because all the features are not there.

Solutions

(1) Inspector Routines. These are public routines written to examine the value of each relevant attribute that otherwise will not be accessible. This may be helpful but unfortunately it is cumbersome to create such classes. Even if these methods are present, they may be defective, making the job of testing that much more difficult.

> *"While abstraction is of potentially great benefit to the modeler, the designer, the user, the maintainer, and the reuser, it is often of hindrance to the defect manager."*

(1) Memento Design Pattern. A variation of the approach above is to design each class as part of a Memento Design Pattern [2].

(1) Encapsulation Breaking Mechanisms. Friend functions in C++ belong to this category. These can be defined to cut across the encapsulation wall of an object. They can then access the internal state of an object, yet have inherent side effects that make their use inadvisable in an OO system.

(2) Inspection. Inspection has proven effective in defect management of partial or distributed implementations. Inspection techniques specialized for object technologies have been developed [3].

(2) Leaf Class Testing. This is a testing technique that evaluates the abstraction structure using its concrete implementations [1, 3].

Encapsulation

In encapsulation, an object is defined as a collection of interrelated concerns wrapped into a logically cohesive unit. In OO, applying encapsulation is not restricted to the composition of classes and objects but also applies at higher levels, for example to form packages and sub-systems.

Problems

(1) Scope Escalation. The routine no longer can be considered the logical unit for testing. That honor now must go to the class. This does not in any way mean that the routines of a class are not tested, but that unit testing in OO must be done in the context of a class.

(2) Hierarchy Integration. How can you test whether the higher encapsulation levels are communicating with each other in the expected fashion?

Solutions

(1) Inspection. Inspection techniques specialized to OO could focus on the algorithmic, initialization, and temporal characteristics of individual routines in a way that is not possible when testing.

(1) Context Testing. This is the idea of giving up individual testing of the individual routines and instead testing them in the context of their operation and collaborations. This is of course risky, as the routines will only be tested in known contexts.

(2) Inspection. Inspection allows evaluation of interaction of packages at integration level.

(2) Multi-Table Class-Responsibility-Collaboration (CRC). This is a simulation technique that allows the evaluation of the design of a system from the perspective of package integration and reduction of coupling [3]. The technique is applied during design evaluation.

Genericity

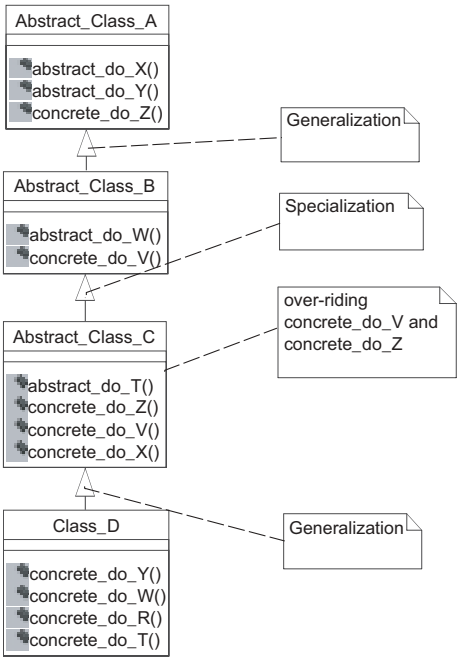
Genericity is the principle of type independence. To facilitate reuse, it is most useful to write components (e.g., classes) that work with a variety of types under a variety of situations. While not all programming languages provide for genericity at the moment, it is likely that its implementation and use would increase in the future. Eiffel [4] is one of the few OO languages that implement this concept effectively.

The rules are quite straightforward. We simply declare the type to be of some generic identifier and then use that same identifier as a placeholder or name whenever a given type is referred to. As such, generic classes are not classes in the strictest sense; they are templates for classes that hold at least one unspecified type. Only when all the unspecified types are *pinned down* does a class emerge. This means that you can write a generic class but you cannot create an object of that generic class. For that to happen, you have to first specify all the generic type placeholders using valid extant types, thus creating a true class. Only then can you create (instantiate) an object. Using genericity, it will then become possible to provide a wide range of libraries of very useful reusable classes such as container (object structure) and graphical user interface classes.

Problem

(1) Type/Behavior Variability. The varying behavior of an object based on the type or combination of types with respect to which it has been instantiated creates a test case explosion.

Figure 1: *Testing Class Hierarchies*



Solution

There is no established calculus here, and problems may emerge from many unexpected corners. No good sure-fire solution exists here. Care, a good deal of anticipation of potential problem areas, and lots of testing with a wide range of potential types is best. Some guidelines (not really solutions) to this end appear in [4].

Inheritance

Inheritance is a kind of relationship between classes. It is one of the central features of object orientation. While it is not necessary to have inheritance in order to have an OO system, most such systems do incorporate inheritance. Inheritance can bring a lot of advantages; the most frequently cited is, of course, facilitating reuse.

A class should implement a particular type A sub-class, therefore, it is best to implement a corresponding sub-type. In other words, our class hierarchy should mirror our type hierarchy. This is usually called generalization. Other forms of inheritance do exist that do not follow this mirror image principle, including specialization and restriction, which do provide particular testing challenges.

Problems

(1) Substitutability Problem. Although a sub-type does satisfy and only strengthens (extends) the preconditions of its parent type, a sub-class does not necessarily do so. As such, a sub-type (generalization) can substitute for the parent class but objects built on specialization or restriction cannot. Such substitutions are, however, among the most common errors in OO.

For example, there is a case of restriction when you take a class such as SIMPLE_INTEREST_ACCOUNT and suppress the interest calculating features of it altogether to sub-class it into the new type NO_INTEREST_ACCOUNT. This new type does not have interest calculating features and thus cannot act as a sub-type of SIMPLE_INTEREST_ACCOUNT although it is a sub-class of it. This creates a problem in testing in that you do not quite know whether to test the suppressed features (as they are still part of the inheritance structure and implementation) or to ignore them (as they are not part of the type being implemented).

Under such circumstances, the testers will have a tendency to look at the contract for the restricted type and then only test according to that contract, leaving behind all the potential side effects of the suppressed features.

(2) Mixing Inheritance Styles. Many designers mix different forms of inheritance in the one-class hierarchy. Although like many of the previous issues discussed, this is ultimately a design issue, it does impact the way you can effectively test a system. In other words, it can contribute to defects in the system and therefore within the scope of our interest, albeit more from a preventive aspect rather than a corrective one.

Imagine the situation depicted in Figure 1: The issue here is that class (D) is a sub-type of (C) and can be substituted for it. Class (C), which may be instantiated (or not), is however not a sub-type of (B), making (D) also not a sub-type of (B). Class (B) is a sub-type of (A), but nothing below it is, even though there might be a lot of further levels. How would you adequately test such a hierarchy?

(3) Deeply Nested Hierarchies. Even generalization, the sub-typing form of inheritance, presents challenges in defect management. In such a hierarchy, the tendency would be only for the leaf nodes to be instantiable. This, however, does not mean that all the features of all of the abstract classes are deferred, far from it. If a class cannot be instantiated, it cannot be tested directly.

Testing a class indirectly must ensure that the class is tested with respect to all possible permutations of the hierarchy down to each individual leaf level that can be instantiated. In a deep hierarchy that is also wide, this creates a combinatorial issue. There are issues, even in the case of a deep but narrow hierarchy. The complete contract of a leaf class is really the union of the contracts of all the parent types, many of them with some implemented operations; it is very possible to miss testing some of them.

(4) Multiple Inheritance. It is possible for a class to inherit from more than one super class directly. Multiple inheritance itself can be of two principal types: simple multiple inheritance and meshed inheritance also known as repeated inheritance. Meshed or repeated inheritance is the case where at least two of the super classes have a common ancestry.

The most obvious issue with testing in a multiple inheritance situation is that a sub-class may inherit a feature with the same name from more than one parent. The child class could use one or the other, but testing with respect to one may

not be adequate when the other is used. The object may interact with other objects, including a shadow or alias of itself with many strange and unexpected consequences, including method runtime clashes.

Solutions

(1) Inspect the Formal Contracts. If each type is written as a contract with its pre- and post-conditions and invariants carefully expressed, it would be possible to easily inspect the contracts of classes in a hierarchy to see if one is a sub-type of the other. There are simple rules that can be applied during an inspection session such as *the precondition of the child class must only extend* or *strengthen the precondition of the parent class* or *leave them unchanged*.

(1) Redesign. All hierarchies based on specialization or restriction can be rearranged into hierarchies of generalization.

(1) Use of Context Testing. This is giving up testing the individual routines individually and testing them in the context of their operation and collaborations. This is of course risky, as the routines will only be tested in known contexts.

(2) Redesign, Avoid Mixing Styles. It is a simple matter of avoiding the mixing of styles during design; you can always convert to generalization (see above).

(2) Segregate Styles. If it is not practical to convert styles, say when you have inherited the code and cannot redesign it, then you should consider each type set as a separate hierarchy and test accordingly. This means that starting from the leaf level, every time the inheritance style changes, all levels below are to be considered (logically abstracted into) one class in a current style relationship with the class above the current location.

(3) Avoid Deeply Nested Hierarchies. One solution is to avoid the problem altogether. As a rule of thumb, hierarchies of more than four to five deep are to be avoided unless they are structurally necessary (e.g., graphical user interface).

(3) Use Flattening Tools. High quality *flattening* tools – those that assist in producing a unified contract by collapsing the contracts involved in a hierarchy – can be of help but the problem is also one of logic and of testing, not of visualization alone.

(4) Avoid Multiple Inheritance. Current advice by many leading practitioners in OO is to avoid multiple inheritance if it is not absolutely necessary (very rarely is it so).

(4) Inspection. Use inspection rather than testing to trace through the logic of multiple inheritance. Again, those inspection systems designed specifically for OO would provide facilities to deal with such issues.

Polymorphism

Polymorphism is the ability of an object to be many forms. A powerful, important, and useful mechanism available in most OO programming environments, polymorphism is considered the ability to substitute one type for another, or in other words bind a reference to multiple instances of different types, and is often closely linked to the concept of dynamic binding. Dynamic binding allows the binding of an object to be deferred to as late as run time, thus permitting the use of different object types, depending on the context.

> *"It is important to realize that virtually every step in the software process is an occasion to introduce a defect that would ultimately manifest itself in the product being constructed."*

Problems

(1) Incorrect Binding in a Homogenous Hierarchy. In homogeneous systems when various methods belonging to a polymorphic structure are closely related both conceptually and operationally, testing might not easily reveal a binding to an incorrect method.

(2) Server-Side Change. A polymorphic server might change without any regard to the client. Under such circumstances, an unchanged client may no longer be able to bind with the server.

Solutions

(1) No Real Good Solution Exists. Extreme care and extensive value testing are to be employed. Some techniques such as evaluating against an explicit post-condition might be helpful but this is not a complete solution.

(2) Inspection. Logic of the binding between the server and the client can be clear during inspection.

(2) CRC. Anthropomorphization through the use of CRCs assists in clarifying the role of the server and its obligations. This is in essence a simulation and is employed during design or redesign. Of course, this technique is ineffective when the server is changed without knowledge of the client side.

Process Issues, Managing Defects

It is important to realize that virtually every step in the software process is an occasion to introduce a defect that would ultimately manifest itself in the product being constructed. Conversely, every step of the software process should be considered an opportunity for defect management. As software engineers, you must consider opportunities to prevent defects and opportunities to detect and therefore remove defects that have already been injected.

Another important realization, however, is that no defect management technique by and of itself is purely a preventive or a corrective one. For example, engaging in design inspection might provide the potential to identify and correct many defects that, if not resolved, would lead to defects in code. From the perspective of the design activity, this is corrective (as you are correcting the design) whereas from the perspective of implementation, it is preventive (as you are preventing the propagation of defects to implementation).

A number of such solutions and the software process stage in which they may be used are shown in Table 1 (see page 16). Software engineers must therefore select and utilize techniques that contribute to production of high quality requirements. These techniques assist in preventing the injection of defects of omission and commission into our specification document. This early preventive treatment has the potential to save you much defect management of the corrective kind later in the process.

Follow this by employing corrective techniques that attempt to identify and help remove defects already extant in the requirements document. Furthermore, you should deal with techniques that concern design, so you may generate defect-free design as much as possible. Designs, irrespective of the effort expended to generate them, will rarely be defect free. We still need to deal with design defect identification techniques such as design inspections. Program code defect identification through testing and

Software Engineering Process Stage	Preventive (P) or Corrective (C)*	Task	Technique
Specification	P	1. Construct Common Dictionary 2. Set Focus/Goal 3. Build Consensus 4. Cover Model Space 5. Cover Functionality 6. Cover Non-Functional Requirements	1. Process Element Dictionary (PED) 2. Quality Matrix 3. State-Behavior Modeling (SBM) 4. UML, Formal Specification (e.g., Object Z) 5. Use Cases 6. Architecture
Specification	C	1. Validate Requirements 2. Verify Requirements	1. Requirements Inspection 2. Requirements Inspection, CRC, Formal Methods
Design	P	1. Ensure Traceability 2. Cover Design Space	1. Requirements Traceability Table (RTT) 2. Architectural Patterns, Design Patterns, Formal Derivation, Contracts
Design	C	1. Validate Design 2. Verify Design	1. Requirements Traceability Check (RTC) 2. CRC, Formal Proofs, Design Inspection, Design Simulation
Implementation	P	1. Ensure Uniformity 2. Ensure Traceability 3. Ensure Design Proximity 4. Ensure Accuracy	1. Coding Standards 2. RTT 3. RTT, Feedback 4. Pair Programming
Implementation	C	1. Defect Identification 2. Failure Detection 3. Integration	1. Code Inspection, Static Analysis, Automated Analysis 2. Dynamic Testing; Specification Testing, Use Case-Based Testing 3. Integration Testing (e.g., Couple Testing, Pair-WiseTesting or Binary Testing), Regression Testing

** Indicates whether the technique in the right column has a preventive or corrective effect on the stage on the left.*

Table 1: *Defect Prevention Techniques*

code inspection will also be needed for the same reason.

Finally you must deal with integration and defect management at the system level. Specific techniques for all these levels are available in the literature [3] and due to space limitations shall not be further discussed here.

Summary

This article presents a fault model for the OO paradigm of software development. This fault model concentrated on specific issues, whether product-based or process-based, that pertained principally to the object paradigm or resulted from its application. In doing so, however, no representations were made in terms of the absence or impossibility of other forms of faults that can arise independently of the paradigm utilized. As such, the model as presented is partial and focused.

The fault model describes the many potentials for producing defective software that might emerge as a consequence of utilizing the OO approach. It also discusses the difficulties that might possibly be encountered in managing and reducing the ultimate defect content of the OO code.◆

References

1. Voas, J. "Object-Oriented Testability." 3rd International Conference in Achieving Quality in Software. Chapman and Hall, 1996: 270-290.
2. Gamma, E., R. Helm, R. Johnson, and J. Vlissides. Design Patterns: Elements of Reusable Object-Oriented Design. Addison-Wesley, 1995.
3. Younessi, H. Object-Oriented Defect Management of Software. Prentice Hall, 2002.
4. Meyer, B. Eiffel: The Language. Prentice Hall, 1992.

About the Author

Houman Younessi is professor of Computer Science at Rensselaer Polytechnic Institute-Hartford Graduate Campus. He is a leading educator, practitioner, and consultant in object technology. Houman is the originator of the Single Building Model methodology and also a key member of the Object-Oriented Process, Environment, and Notation (OPEN) consortium and one of the designers of the OPEN Process. He is author of three books, including "Object-Oriented Defect Management of Software." Younessi has been instrumental in the formulation, evaluation, and promulgation of the ISO 15504 Software Process Improvement and Capability Determination standard for software process capability measurement. Younessi is an international speaker, and has spoken at the International Conference on Software Engineering, the Conference on Technology of Object-Oriented Systems USA, and the Asia-Pacific Software Engineering Conference.

Department of Computer Science
Rensselaer Polytechnic Institute
Hartford Graduate Campus
275 Windsor St.
Hartford, CT 06074
E-mail: houman@rh.edu

Defect Management Through the Personal Software Process

Iraj Hirmanpour
AMS, Inc.

Joe Schofield
Sandia National Laboratories

Software quality improvement begins with defect-free software. The Personal Software Process℠ (PSP℠) defect management framework provides individual software engineers with the tools to prevent and remove defects early in the life cycle. Our experience with the PSP indicates that the application of discipline methods such as PSP provides a mechanism for defect prevention as well as early defect removal and substantial reduction in test time. In this article, we describe the PSP defect management framework and quantitatively demonstrate the reduction of defects by using the PSP defect management methods

Metrics here, metrics there,
metrics metrics everywhere.
ERA and GPA, MPH and MPG;
LDL and HDL, UCL and LCL,
RPM and RBI, BPS and DPI,
upper limits, lower limits,
in-bounds, out-of-bounds,
on schedule, on budget,
out of scope, out of hope!

Can there be any doubt that metricssurround us [1]? Measurement and metrics are foundational for understanding an engineering process. In the software-engineering world, the collection of metrics has been problematic, yet its need persists for process improvement and product quality monitoring. Project measures that predict cost and schedule are easier to obtain and are widely used. However, collecting software defects to measure quality is more difficult and thus not as pervasive as other project measures.

The key measure related to software quality is, of course, defects. According to Watts Humphrey, developer of the Personal Software Process℠ (PSP℠), "The defect content of software products must first be managed before other more important quality issues can be addressed" [2]. Any organizational claims that software quality is improving are unreliable sans defect measures. This article focuses on the PSP defect management system, and reveals how a systematic approach to defect collection and analysis provides individual engineers with the ability to remove defects early in the software development life cycle.

Personal and peer reviews are primary sources of defect detection. Test results are another source of defect detection, albeit a more resource intensive activity. Worse yet, change requests and *trouble reports* are evidence of defects that have made their way to the customer. The PSP's focus on quality software products ameliorates the collateral damage associated with defects discovered by the customer. Despite large investments in testing strategies, the average U.S. software defect removal rate is about 85 percent [3]. These dismal results are the consequence of using less disciplined software-engineering practices that rely on code and test cycles to remove defects.

Given that software engineers inject defects, they should be responsible for identifying and removing them. Our experience with the PSP, supported by the Software Engineering Institute (SEI), indicates that the application of disciplined methods such as PSP reduces the number of defects injected in the process and the amount of test time required to detect and remove them. This reduction is achieved primarily by lowering the number of defects that are introduced and secondarily, by removing defects early in the life cycle rather than in testing. Using the PSP defect management framework, this article will demonstrate how software engineers can improve their defect management process.

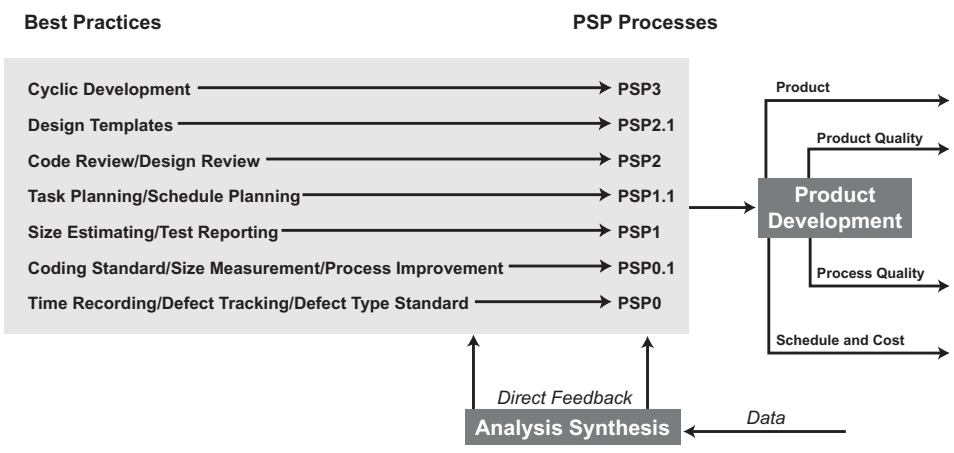
The PSP Framework

The PSP framework is a data-driven feedback system that allows individual software engineers to continuously improve their personal processes by applying statistical process control techniques at the individual level. A PSP practitioner uses a defined software process to apply a set of practices to develop products, while collecting data as part of the development process. Figure 1 illustrates how the collected measurements are used to analyze and assess the impact of a practice on the product and/or process using a feedback loop. This feedback becomes an inherent part of all future product development processes. The framework therefore, offers a road map for collecting data. By analyzing the data, engineers are able to modify their practices and thus improve predictability and quality.

The framework depicted in Figure 1 shows the seven process steps numbered from PSP0 to PSP3. On the left are the new practices that are introduced at that process step. In PSP 1.1 process step, for example, task planning and scheduling planning practices are introduced. It is important to notice that all previous process steps evolved into the schedule planning and tracking process of PSP 1.1. In other words, process steps are evolutionary and cannot be skipped.

A PSP practitioner decides to use one of these process steps to produce software artifacts. The SEI recommends using practices embodied in PSP 2.1 that

Figure 1: *The Personal Software Process Framework*



℠ Personal Software Process and PSP are service marks of Carnegie Mellon University.

Defect Types	
10	Documentation
20	Syntax
30	Build, Package
40	Assignment
50	Interface
60	Checking
70	Data
80	Function
90	System
100	Environment

Table 1: *PSP Defect Types*

inherits all previous practices. The product development process of a PSP practitioner, regardless of which process step is employed, produces two classes of output: the project product and a set of metrics on process and product. Contrast this approach with the classic code and test approach that emphasizes the maturation of the product by removing defects (or discovering requirements) during the test phase.

The PSP Defect Management

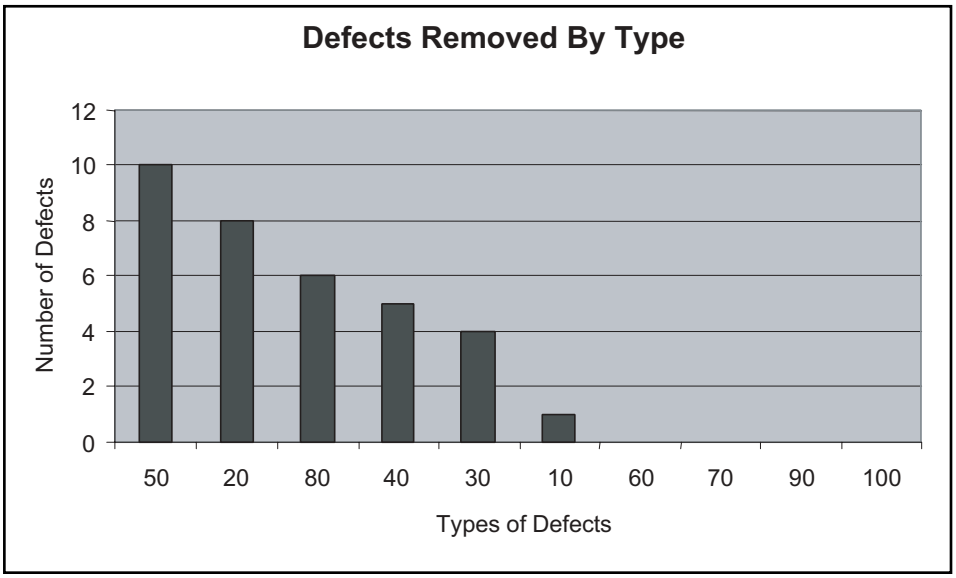
The goal of any defect management process is to eliminate defects from software products. Unfortunately the practice often merely tends to reduce defects [3]. The PSP framework promotes defect management. While learning and practicing the PSP, engineers are required to collect and record data on the process and product during development, including defect data. Starting with the first PSP process step, engineers are introduced to a defect collection method. Within the PSP, a defect is defined as anything that will result in failure of software to operate, causing rework to correct it [4]. The defect collection method consists of establishing a defect classification scheme and recording defect attributes.

For each defect identified, engineers record the defect type based on the classification scheme in Table 1, as well as the following: the injection phase, removal phase, and correction time [5]. To improve a process, it is necessary to know the current state of the process. By writing a program using the first PSP process step, engineers gain insight into their process by exploring the question, "What is my current defect injection/removal rate?"

This defect collection process continues until the fifth step of the PSP during which students would have written seven programs. Once sufficient defect data are collected, engineers are required to determine defect injection and removal rates, and the associated correction time. Armed with this information, engineers produce a design review checklist and a code review checklist based on their personal defect profile. A typical defect profile created by PSP defect data is shown in Figure 2.

Furthermore, the PSP framework provides a structured review process that the engineer follows using the checklist to review his or her work. In the example data shown in Figure 2, the engineer will have defect types 50, 20, and 80 on the checklist because according to personal data, they are injected most often and consume the largest repair time. The fifth process step of the PSP (PSP2) introduces the design review and code review activities as part of the process. The goal is to remove all defects before compiling and testing.

Figure 2: *Defect Profile of a PSP Student*



The PSP review framework consists of process scripts, checklists, and time and defect collection forms. The engineer follows the review script that specifies three phases: review, correct, and check. For each item on the checklist, engineers review each line of design or code from beginning to end. Each time a defect is found it is corrected and its correctness is verified. Time spent fixing the defect and the type of defect are recorded in the defect log. The PSP review process, therefore, is a structured and measured process. The collected review data includes the time spent in review, the number of defects found, the time spent fixing defects, and the number of lines of design or code reviewed.

From these measures, you can derive metrics such as lines of code (LOC) per hour reviewed and defects detected per hour. During the post-mortem phase of the project, two additional metrics are derived called *yield* and *appraisal to failure ratio* (AF/R). Yield is defined as percent of defects removed before the first compile. AF/R is defined as the ratio of percentage of the total time that engineering spent reviewing a product (appraising) and percentage of time that engineering spent compiling and testing a product (correcting failures). An AF/R ratio of two reveals that twice as much time was used to review the product compared to compiling and testing it.

Data gathered during the PSP class is then used to assess the quality of the review and to develop an improvement strategy. Some of the PSP historical data suggests that a review rate must be less than 200 LOC per hour, the yield goal should be around 80 percent, and the AF/R should be greater than the number two.

Unfortunately, the data collected on current practices of software engineers indicates that the opposite is true. Engineers prefer to rush through the coding phase with little or no design, minimize reviews, and then correct defects during the compile/test phase.

The PSP Class Defect Data

As described earlier, PSP students develop 10 programs following progressively evolving practices using the PSP while collecting data on their work. The first seven programs are written using planning, design, code, compile, test, and post-mortem as process phases. Although activities within each phase grow in sophistication, phases stay the

same until program eight, at which time quantitative management practices are introduced and two new phases – design review and code review – are introduced. The expanded and complete PSP consists of the six process phases listed earlier with an additional review phase following both the design and code phases respectively.

The first program is written using the PSP0 process to establish a baseline of current state. Table 2 shows defect data for five PSP classes. Students attending these classes are practicing engineers; all are college graduates. Fifty percent of the students have a master's degree and an average of 11 years experience. As depicted in Table 2, the range of defects varies from 69 to 124 (variation of 55 percent) among classes with an overall average defect rate of 100 per thousand lines of code (KLOC). Similarly, the test defect range varies from 25 to 55 (variation of 45 percent) for each of the five classes and contains an average test defect of 38 defects per KLOC. Data from PSP classes consistently shows a wide variation in performance among software engineers. Variation in defects is no exception.

These data form the baseline from which performance improvement is measured. In addition to helping engineers, this data is also useful to the organization. If this organization were suddenly required to estimate defect injection rate as part of preparing a quality plan, 100 defects/KLOC would be a valid estimate based on historic performance. In lieu of these measures, the organization is void of quantitatively determining its defect profile or the amount of time engineers use to fix the *bugs*.

Once all of the defect management practices are introduced, a sharp drop in both total defects and test defects is achieved. As shown in Table 2 in all classes, the overall average in process defects improved by 50 percent and overall average test defect improved by 63 percent. Since testing removes only a fraction of defects [2], fewer defects discovered in test, while performing similar levels of defect removal, equates to fewer defects in the final product. Measured quality improvements are an additional benefit of following these process steps.

The next question is, "Do engineers who learn and apply PSP in their work processes produce higher quality products than non-PSP trained engineers?" To answer the question, three recent graduates of a PSP class agreed to collect and share data on their projects based on the PSP model. They gathered data on 13 small maintenance projects with a total of 13,914 LOC. As shown in Table 3 on those 13 projects, the total defects per KLOC were 22 and test defects per KLOC were reduced to four.

	Number of Students	Defects Per KLOC Start	Defect Per KLOC End	Test Defect Per KLOC Start	Test Defect Per KLOC End
Class 1	8	69	40	25	9
Class 2	7	108	28	40	11
Class 3	10	83	24	33	21
Class 4	7	124	74	35	10
Class 5	11	119	83	55	17
Average		**100**	**50**	**38**	**14**

Table 2: *Comparison of Defect Profile at Start and End of the PSP Class*

While not a statistically viable study, the similarity between the results over five classes and those from 13 actual projects based on the PSP model reinforces the fact that dramatic improvement can be achieved if graduates continue to follow the PSP process.

Summary and Conclusion

Software quality begins with the removal or substantial reduction of software defects before other quality attributes such as maintainability, portability, reliability, or usability can be considered. A defect is referred to as anything that causes the software not to function as specified and requires efforts to correct it. Needless to say, a major source of software defects is missing or incomplete requirements, which are not addressed in this article and relate to the requirements engineering process.

However, once a set of requirements is agreed upon, the next challenge is to design and build software that satisfies the requirements and is defect free, that is, it functions as specified. Once the requirements are specified, defects enter the product during design and coding phases. Since software engineers are engaged in the design and coding activities during which defects are injected, they should also remove them. The quality principle of *do it right the first time* stipulates that these defects be detected and removed by the engineers while in development and not during test or deployment.

The PSP defect management framework enables software engineers to prevent defects and then to identify and remove injected defects early to avoid costly corrections later in the life cycle. There are two components to defect management: defect prevention and defect detection. The PSP defect data collection system provides the necessary information to use statistical methods to identify the root causes of defects and to develop strategies for preventing defect injection. The PSP's structured and measured review process enables software engineers to detect and remove defects early. The measurements taken during the review are analyzed to improve the efficiency of the review process, thus providing a continuous improvement mechanism.

Our experience with teaching classes and collecting data on students supports the notion that as engineers use PSP defect management practices, their defect injection rate is reduced substantially (defect prevention), and defect removal efficiency (defect detection) is increased resulting in reduced test and repair time. Lower costs and higher customer satisfaction follow naturally.

Metrics abound in the construction of software, as in other engineering disciplines. We have attempted to demonstrate how the use of metrics, in this case defect collection and analysis, contributes to measured improvement in software quality and a reduction in development and support time. Addi-tional benefits accrue to organizations as their software engineers continue to practice the PSP as part of their daily activity. The PSP provides a framework for software process improvement. Its processes can sustain

Table 3: *Average Defect Rates*

	Start of class	End of class	After class
Total Defects/ KLOC	100	50	22
Test Defects/ KLOC	38	14	4

enhanced practices within an organization's software engineering community long after the class has concluded.◆

References

1. International Function Point Users Group, et. al. IT Measurement: Professional Advice from the Experts. Addison-Wesley, 17 Apr. 2002: 221.
2. Humphrey, W. S. A Discipline for Software Engineering. Addison-Wesley, 1995.
3. Jones, Capers. Software Quality. International Thomson Computer Press, 1997: 400.
4. Humphrey, W. S. A Discipline for Software Engineering. Addison-Wesley, 1995: 12.
5. Ibid: 44.

Additional Reading

1. Khajenoori, S., and I. Hirmanpour. An Experiential Report on the Implications of Personal Software Process for Software Quality Improvement. Proc. of the Fifth International Conference on Software Quality, Austin, TX, Oct. 1995 <www.sei.cmu.edu/tsp/recommended-reading.html>.

About the Authors

Iraj Hirmanpour is a principal of AMS, Inc., a software process improvement firm and a Software Engineering Institute Personal Software Process℠/Team Software Process℠ (PSP℠/TSP℠) transition partner. He is a SEI-certified PSP instructor and TSP launch coach. Hirmanpour is also a visiting scientist with the Carnegie Mellon Software Engineering Institute collaborating on the transition of PSP and TSP into academic curricula.

AMS Inc.
421 7th St. NE
Atlanta, GA 30308
Phone: (386) 405-4691
E-mail: ihirman@earthlink.net

Joe Schofield is a technical staff member at Sandia National Laboratories, a multi-program laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy. He chairs the organization's Software Engineering Process Group, is the Software Quality Assurance Group leader, and is accountable for the introduction of the Personal Software Process℠ and the Team Software Process℠. He has dozens of publications and conference presentations in the software engineering realm and has taught graduate level software engineering classes since 1990.

Sandia National Laboratories
MS 0661
Albuquerque, NM 87185
Phone: (505) 844-7977
Fax: (505) 844-2018
E-mail: jrschof@sandia.gov

Defect Management in an Agile Development Environment

Don Opperthauser
AgileTek

Agile development practices are sometimes thought of as an undisciplined approach to software development, lacking such things as effective defect management. However, agile development does not hinder the use of formal defect management processes in any way. On the contrary, agile development does much to reduce the incidence of defects in the first place. This article will paint the picture of defect prevention and management within an agile development environment.

There are a number of methodologies and approaches to agile development. For the sake of this article, the discussion will center on how we at AgileTek handle defect management within the context of a software project that uses our Agile+ methodology[1]. There is sufficient overlap between the various agile methodologies that this discussion should have ample application to any of them

Software defects only exist if, at the end of the day, someone says, "This software is not accomplishing the purpose for which it was written with the accuracy, efficiency, and ease of use that was intended."

This article discusses defect management in two broad categories: requirements defects and implementation defects. The term requirements is used in a broad definition to include all types of requirements, functional specifications, and other means to define what the software is supposed to do and, from a functional perspective, how it is supposed to do it. Implementation defects refer to defects in architecture, design, coding, installation, or any other aspect of the technical implementation of a software development project.

Let me begin with a case in point. More than a decade ago, I and the other future AgileTek co-founders received a functional specification from a large ($13 billion today) consumer products company. It was not an overly complex system, but the functional specification ran to more than 400 pages. The painstaking detail of the document was impressive; every detail of the user interface, validation rules, and exactly how everything was to work was all spelled out. We got the job.

While the software was intended for use by the field sales force, our customer was the information technology (IT) organization. We suggested that perhaps it would be wise for our development team to sit down with some of the intended users and review the specifications. We were told that the IT folks had already done that and, moreover, the effort had taken up more of the users' time than they wanted to give; there was no need for any further review. All we needed to do was to build and test the software to spec – what we call *spec conversion* in our business.

What seemed like a straightforward task of turning the specifications into bits and bytes got complicated when we discovered that what it said on page 83 contradicted what it said on page 183 and so forth. Could we have possibly analyzed, absorbed, and understood those 400 pages well enough to catch such problems before we began? Of course not. More importantly, do you think that anyone in the sales force really analyzed, absorbed, and understood those 400 pages even though they approved the specification? Most assuredly not! Their eyes probably glazed over around page 20, and they had no choice but to approve a specification they neither had the time nor skill to understand.

> *"There is no substitute for adequate client involvement. Clients must invest the right amount of time from the right people if they are going to get an effective result."*

Eventually the questions were all answered (by the IT folks, not the users), the system was installed in a test environment, and a group of users came to town for user acceptance testing. At the end of the first day of explaining how the system worked with the users and letting them get some hands-on experience, my architect asked the user-group's supervisor what she thought of the software. "This isn't the software we need!" was her disheartening reply.

I relate this story to underline the importance of *building the right software*.

Requirements defects of any nature are the most disastrous and costly. How are requirements and requirements defects managed in Agile+, a software development methodology[1]? Several of the components of this methodology speak directly to this issue.

Customer at the Center of the Project

One of the problems in the anecdote above was the fact that the people who really needed the system to do their work were isolated from the people who were building the system. Some development organizations try to keep the customer at arm's length. By building a lot of customer involvement into our projects, we ensure that we are getting adequate and frequent feedback to keep the project on target.

In Agile+, the customer is treated as a full-fledged member of the development team with access to all the information to which the rest of team is privy (e.g., defect logs, issue lists, etc.). Once on the team, constant effort is made to ensure that the customer is an integral part of that team. An Agile+ project is steered by a dedicated individual (customer or customer proxy) who is empowered to determine requirements, set priorities, and answer programmers' questions as they arise.

This is one of the most critical issues in managing requirements defects. *There is no substitute for adequate client involvement.* Clients must invest the right amount of time from the right people if they are going to get an effective result.

Flexibility to Meet Client's Special Needs

If there is any conflict between the products produced by our methodology and the customer's needs, the methodology is adapted to serve the customer. For example, Agile+ takes a minimal approach to documentation – only enough to ensure proper execution and maintenance.

However, in regulated environments such as pharmaceutical research, painstakingly detailed documentation is almost always a required byproduct of any related software development. In such a case, normal documentation procedures are modified to meet project requirements.

Business Process Analysis

A thorough understanding of the business objectives that the software must achieve is crucial to reaching desired results. Too often the development team does not get involved early enough in the process to define the software to be built. The discussions that take place during iteration planning sometimes are not enough to ensure an understanding of the underlying business process to be supported. By the way, I am using the term *business process* in the broadest sense, to include manufacturing processes, military processes, or any process that needs to be carried out to accomplish the goals of an organization.

A formal, facilitated business process analysis (BPA), *with the entire development team present*, should begin any development effort. It is usually not enough for some BPA professionals to work for days and weeks to produce a BPA document, hand it to the development team, and say, "Read this." The discussions and nuances that occur during the BPA sessions cannot all be put into words, and certainly the development team cannot gain the depth of understanding needed to design and build the right application without personal participation in the BPA.

There is a very important point behind all of this. Despite all of our processes and technologies, software development is a rather new industry compared to something like building houses, which we have been doing for thousands of years. There are too many variables in building software, too many nuances, and too many possible user actions and paths through the system. Time to gain a personal understanding is needed, and the BPA is the perfect vehicle.

User Stories and Story Actors

Expressing requirements in terms that everyone can understand goes a long way to ensure you are building the right software. Many approaches to requirements definition produce results that are all but incomprehensible to the customers who really understand what the software needs to be.

In Agile+ and many other approaches to agile development, requirements for the system are gathered through user stories (sometimes referred to as use cases) that are developed through customer interaction. A user story does not fully define a requirement; rather, it defines an underlying business need from which the requirements can be determined. Later during architecture development, these stories inform the scenarios that are used to help validate the architecture.

We have added to the concept of *stories* the concept of story *actors*. Actors are personifications of the various categories of users that the system will encounter. Thinking of the requirements in terms of actors brings the requirements to life, and unmasks nuances that would otherwise remain invisible to both the developers and the customer. It enables the requirements to be written in terms of how the system will be used versus desired functions. Finally, by associating who is doing what, it helps conceptualize and compartmentalize the functions.

> *"The key tenet in all agile software development methods is iterative development and the unforgiving honesty of working code."*

This approach allows high-level requirements to be expressed in terms understandable to users who really know what the system needs to do and to executives who must approve them.

Iterative Development

Short iterations allow the customer to see completed functionality very early on so that feedback is not only meaningful, but also received in time to keep the project on track with respect to the final project goals.

The key tenet in all agile software development methods is iterative development and the unforgiving honesty of working code. The concept of iterative development has been around for a long time and is perhaps best known through the application of spiral development.

Our iterations are kept short, generally no more than three to five weeks. Iterations begin with an iteration planning session during which the customer and project team select the user stories to be implemented during the iteration. Once the user stories are selected, the iteration planning continues with consideration of such elements as screen designs, user workflow, data input/output, etc. This is then input to a period of design *(days of design)* wherein business analysts and developers work together to develop specifications and produce component designs. Tests for these designs are developed prior to writing the code; the code is then exercised using these tests.

At the end of each iteration, we deliver working code for the stories implemented and review it with the customer. This enables our customer and us to evolve our understanding, challenge assumptions, and make informed choices and decisions. Using the information gained during the iteration review, we are in a much better position to plan the next iteration.

A software development effort meeting its requirements is analogous to a projectile hitting its target. In effect, each iteration is an opportunity to provide the development project mid-course guidance. By keeping the time period between iterations to no more than five weeks, the feedback loops are kept short, thus providing frequent guidance and ensuring the project never gets far off-track. *Iterative development is perhaps the single most important vehicle for managing requirements defects.*

In addition to building the right software, i.e. effectively managing requirements and requirements defects, the more traditional concept of defect management, *building the software right*, must also be addressed. Implementation defects are a major source of project trouble in traditional methodologies where late-stage integration brings system modules together near the end of the project. This usually results in an unbelievable number of defects. The development team goes into near paralysis while they try to get their newly integrated, defect-ridden system repaired to the point where meaningful system testing can even begin.

One of my colleagues for many years liked to refer to "Larry's two-phase software development methodology – defect creation and defect removal!" Agile+ takes a two-pronged approach to implementation defects. Some of the practices help prevent defects from ever getting into the software, and others facilitate early detection and repair.

Let us examine the practices of Agile+ that impact implementation defects.

Identifying System Components and Interfaces

Clearly defined components and interfaces are key to quality code. Especially for complex systems, it is important to assure conceptual integrity in the final product. Also,

because complex systems can be large, it is important to enable the system to be developed in an environment of distributed ownership.

Architecting a system simply means identifying the constituent components of the system and defining the interrelationship(s) between them. The best architectures are isomorphic (one-to-one) mappings between problem and program space. This ensures that a system's underlying structure and components mirror the problem being solved. This means that for the program to change requires that the problem changes, and as a result, you are *change-proofing* your program. While there may be more efficient ways to solve a problem (e.g., creating one module to perform similar functions by invoking it in a context sensitive way), this efficiency will almost always come at the expense of time spent debugging and later modifying the program if one or more of the functions change.

However, it also means something more. By defining the relationships between the various components, you have gone most of the way toward establishing agreements for the interfaces. The power of interface agreements is that they serve as restrictive liberators. In other words, the individuals working on various system components are free to design the internals of those components without regard for potential untoward effects on the rest of the system – so long as the interface agreements are honored.

As you can see from this discussion, a rigorous approach to identifying components and adherence to well-defined interfaces severely limits the effect that defects can have, thus making it easier to localize and repair defects when they do occur.

Collective Ownership

The team approach leverages the entire team's thinking on critical problems and ensures that no one is working in a vacuum, possibly going off in the wrong direction. The pride of ownership diffuses through the entire team, creating a high degree of motivation to write good, defect-free code. Peer pressure is very effective if there is someone on the team who is creating more than his or her share of defects, thus creating problems for everybody.

Continuous Integration

Software development history is strewn with projects that failed at the critical juncture of integration – bringing all of the components together near the end of the project. Continuous integration uncovers integration issues early. In this manner, integration defects, if any, are introduced one at a time as small pieces are integrated into the system and therefore are resolved more easily.

Relentless Testing/Automated Contract and Regression Testing

Requiring developers to submit virtually defect-free code to start with ensures not only a high quality product, but also consistent quality throughout the project. Agile+ requires that software *contracts* be written and automated tests designed before coding begins[2]. Contracts define the pre-conditions, post-conditions, and class invariants for any function to be written, and the automated tests check for these.

> *"Too often defect management is so focused on defect repair and getting the software out the door that we fail to learn from what is happening."*

Before a developer can check code into the configuration management system, he or she must have a build of the entire system, including new code on either his or her development computer or on a test system designated for that purpose. The developer must then run not only his or her newly written automated test for the new code, but also all automated tests that exist for the entire system. Only when all tests return defect-free results may the developer add his or her new code to the project. In this way, very few defects are introduced into a project build and the system under development is maintained in a relatively defect-free state at any given time.

Refactoring[3]

Designs and architectures are boldly changed when needed to maintain the correct architecture throughout the project. Development that proceeds without fully automated tests on the entire system as described above soon reaches the point where major changes in architecture become too risky. Developers then will use workarounds, *kludges*, and other poor programming practices to avoid doing what they should do – make the major changes necessary to make the system work the way it really should. Refactoring is what enables this architectural and design rework. It keeps the system clean and contributes greatly to minimizing defects and making it easier to identify and repair defects that do occur.

Pair Programming

"Two heads are better than one" (and sometimes cheaper, too). Putting two developers on very complex or high-risk tasks decreases the risk of poor results.

Coding Standards

The maintainability of code is directly affected by having good coding standards, not the least of which is guidelines for properly commenting code.

In addition to these best practices designed to prevent defects, you will of course need some system for tracking defects, defect repairs, certification of repair after retesting, documentation of items found in system testing that are not really defects but future enhancements, etc. These tracking systems may be more or less sophisticated depending on project complexity and client requirements.

In some regulated environments, such as pharmaceutical or Department of Defense environments, it may be necessary to track defects and repairs back to the original affected requirements. The goal of defect tracking in Agile+ is to have no more tracking than necessary to achieve project goals, legal or client requirements, and metrics desired to analyze effectiveness of the software development effort.

Defect management systems should track a number of basic things, including the following:

- An accurate description of the defect, including detailed steps for reproducing the defect and as much information as possible about the application environment at the time the defect was discovered.
- History of the defect, including who discovered it, who is assigned to repair it, when it was fixed, who is assigned to verify and certify the repair.
- Where the defect originated. In other words, why is this defect here? Is it a mistake in requirements, architecture, design, coding, or perhaps faulty tools such as a compiler defect, etc?

Too often defect management is so focused on defect repair and getting the software out the door that we fail to learn

from what is happening. Analysis should take place to determine why the defect occurred, not just where. Was it bad information, inadequate skills to do the job right, careless execution, or some other cause? Knowing why the defect occurred will help us continuously improve our processes and performance.

In conclusion, Agile+ provides a set of practices that focus on prevention of both requirements and implementation defects while facilitating the effective and efficient identification and repair of defects that do get into the project.◆

Notes

1. Agile+ is a further refinement of Code Science, which is described in "Odyssey and Other Code Science Success Stories," CrossTalk Oct. 2002: 19-21.
2. Bertrand Meyer introduced the idea of Design by Contract. For more on this, see his book Object-Oriented Software Construction. 2nd ed. Prentice Hall, 1997.
3. Martin Fowler defines refactoring as "the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure." For more on refactoring, see his book Refactoring: Improving The Design of Existing Code. 1st ed. Addison-Wesley, 1999.

About the Author

Don Opperthauser's software development career has spanned responsibilities from head-down programmer to senior executive responsible for multi-million dollar projects for Fortune 100 companies. He successfully managed a multi-million dollar software project for the third largest U.S. corporation. The project, which was critical to the client's product development, was completed in a record five-month period. Software developed under his leadership was key in providing the enabling software for the most profitable business unit of another Fortune 100 Company.

Phone: (847) 770-1637
Fax: (847) 813-4903
E-mail: dopperthauser@agiletek.com

WEB SITES

Software Certifications

www.softwarecertifications.com

The goal of Software Certifications is to offer an independent professional certification that carries weight in the information services marketplace, and is considered valuable to all of those professionals who seek and attain one of the certifications. The Quality Assurance Institute Professional Certification division sponsors and administers the software certification programs. Available certifications include the Certified Software Quality Analyst and a newly added Certified Software Project Manager.

Institute of Configuration Management

www.icmhq.com

The Institute of Configuration Management is known for its CMII process, which is an advanced version of configuration management (CM). CM is the process of managing products, facilities, and processes by managing their requirements, including changes, and assuring conformance in each case. CMII is CM plus continuous improvement in these five areas: (1) accommodate change, (2) accommodate the reuse of proven standards and best practices, (3) assure that all requirements remain clear, concise and valid, (4) communicate (1), (2) and (3) promptly and precisely, and (5) assure that the results conform in each case. CMII expands the scope of CM (beyond design definition) to include any information that could impact safety, quality, schedule, cost, profit, or the environment.

International Society of Six Sigma Professionals

www.isssp.com

The International Society of Six Sigma Professionals (ISSSP) exclusively promotes the interests of Six Sigma professionals. It is a global community comprised of individuals seeking to learn how Six Sigma might be introduced – or integrated – into their business processes, deployment and implementation experts, and businesses that are implementing Six Sigma and other change management practices. ISSSP is committed to the advancement of education, research and implementation of the Six Sigma methodology.

Lessons Learned From Another Failed Software Contract

Dr. Randall W. Jensen
Software Technology Support Center

Software project failure has been with us for a long time. Volumes have been written about the list of potential problem areas in the acquisition of large, complex software systems. The list includes simple things like the cost of reuse, the acquisition process, unrealistic expectations, and the development environment. The list has not changed much in the last 30 years. Unrealistic cost and schedule estimates are causes for project failure as often as inadequate technology. Source selection is a critical acquisition process step. Proper preparation and diligence in this step is key to a successful software project. There are several activities essential to successful project planning and acquisition, including risk assessment. This lessons-learned discussion is based upon a post-mortem analysis of an avionics software development. The intriguing analysis results show this project was neither unique nor abnormal. The problems that surfaced during the project's inception and following downhill plunge were common in the mid-80s environment and are still common today. The purpose of this discussion is to highlight the major software development and management issues that led to this project's failure. The issues presented here are timeless; that is, they are as likely to arise today as they were at any time in the past.

One of the most dominant and serious complaints arising from the ongoing software crisis is the inability to estimate with acceptable accuracy the cost, resources, and schedule required for a software development. Traditional intuitive estimation methods have consistently produced optimistic results that have contributed to the too familiar cost overrun and schedule slippage.

Several schedule and cost estimation methods have been proposed over the last decade with mixed and partial success due, in part, to capability and stability limitations of the estimation models. A significant part of the estimate failures can be attributed to a lack of understanding of the inner workings of the software development process and the impact of that process on parameters used in the schedule and cost estimates.

For example, a major avionics modernization program was started in the mid-1980s. The development contract award for the system development was issued, but by 1990 it was apparent the software product would not be delivered. The government accepted the incomplete software and completed the software in-house. The failure of another software development is, by itself, not noteworthy.

Unfortunately this example is very common. Industry software delivery statistics are quite dismal. Fifty percent of commercial software products are delivered over schedule, 33 percent are cancelled, and 75 percent are operational failures. Government software delivery statistics are similar.

The following lessons learned discussion is based upon a post-mortem analysis of this avionics software development. The intriguing analysis results show this project was neither unique nor abnormal. The problems that surfaced during the project's life were common in the mid-1980s environment and are still common today.

The purpose of this article is to highlight the major software development and management issues that led to this project's failure. The issues presented here are timeless; that is, they are as likely to arise today as they were at any time in the past.

Lessons Learned

This analysis was a classic study of projects gone awry. There are many lessons that can be extracted from the contract history. Since my experience is largely centered on the relationship between software development and methods for predicting cost and schedule, I focused my attention on the development environment impact on the cost and schedule of the avionics program software. I will not touch upon other areas such as risk management that contributed to this project's failure.

There is no implied order of importance to the lessons enumerated here. Each of these issues contributed significantly to the software development failure. Taken together the issues spelled disaster.

Software Reuse and COTS

The magic elixir reuse was the solution to the industry's software cost and schedule problems in the 80s. That was a time when the new programming language Ada and the concept of reusable software component libraries were very popular. Reused software in a mid-1980s development equated to *free* software much as commercial off the shelf (COTS) software does in a development environment today. Unfortunately, software component libraries never became widely available, and the cost savings associated with reusable software were not as large as predicted.

The concept of COTS software is easiest to understand through a *black box* analogy. A COTS component is a black box that can be fully utilized with no knowledge of the box content. White box behavior, on the other hand, requires some knowledge of the internal box workings. A software component is a white box when (1) modification is required to meet system requirements, (2) the component reliability is in question, or (3) the knowledge of the component and its documentation are inadequate for the application. When the white box condition occurs, the effort to implement the software system must be increased to account for reverse engineering the component, coding the component changes, and additional testing required to assure proper component performance after the modification.

The reusable software baseline proposed for this avionics system development was in development by a competitor for this project. The competitor's system had a different architecture and different operational and performance requirements. The delivery schedule for the baseline system that contained the reusable software was from six months to a year following the start of development for the proposed avionics system.

The contractor defined about 90 percent of the *existing* avionics system soft-

ware as reusable with only about 10 percent of the source lines to be developed as part of the modernization program. The assessment was made with only high-level design information from the baseline system. In reality, there was little reusable software available for the program. Even if the existing software had been available at the start of development, the new software requirements for the system would have precluded any benefit from reuse. The baseline software was not being developed with reuse as an attribute, nor could its developer have been expected to be more than minimally cooperative with the adaptation of that software to the new requirements.

Lesson 1: Reusable (COTS) software never was, is not now, and never will be free.

There is always some development effort expended in the use of reusable software components to engineer and integrate those components into a software system.

Proposal Evaluation

Source selection is a critical step in the acquisition process. Proper preparation and diligence in this step is key to a successful software project. There are several activities essential to successful project planning and acquisition, including risk assessment. This analysis focuses only on the schedule and cost estimate evaluation.

Proposals are typically divided into technical and cost portions. The technical proposal is carefully analyzed and evaluated by a team of application and technology experts. A second team of financial experts evaluates the cost proposal. There are two potential problems with this two-team structure. First, the two teams often perform the evaluations independently. Technical risks that impact the cost estimate are not communicated adequately, as happened in this project.

Second, the cost evaluation team is often relatively inexperienced in using software estimating methods and tools. This does not mean the team is inexperienced in financial and accounting methods. Software estimating is a specialty that requires training and experience. Training for this discipline is typically little more than keyboard training; that is, "What key do I press to get a cost profile?"

It appears the technical and cost evaluation teams were working independently during the proposal evaluations on this source selection. The reuse issue created by the overlap between the modernization program and the reusable software development should have been a major concern. The high reuse level, or extremely low size estimate, was obviously a key in the contractor's proposal strategy. Neither the cost evaluation team, nor the technical team, questioned the high reuse percentages. The teams also failed to be concerned about the low size estimate.

Lesson 2: Technical proposal evaluation should be tightly coupled with cost and schedule evaluation. Isolation of the two activities leads to contract disaster.

A *should-cost* estimate should be completed prior to the source selection phase to establish a project plan and provide the cost evaluation team with a sanity check for the upcoming proposal evaluation. The sanity check will vary considerably as contractor capability and risk assessments are refined during source selection. The cost evaluation team should provide the should-cost estimate.

A technique to strengthen the sanity check is through using an independent third-party estimate. This type of estimate is frequently requested by the acquisition team to validate and refine the cost team estimate.

Estimating Practices

It is important to develop a reasonable estimate at the outset of any software acquisition. The estimated cost and schedule projections are vital for proper project planning, source selection, resource management, and risk management. The absence of a valid estimate is a primary cause of cost and schedule overruns, programs that spiral out of control, and failed programs. Estimate importance is often ignored or minimized in the rush to get the project underway.

A significant part of estimate failures can be attributed to a lack of understanding of the inner workings of the software development process, and the impact of that process on the parameters used in the schedule and cost estimates. One of the poorly understood variables in the development process is the impact of management on the ultimate cost and schedule of the delivered product. The style and environment imposed by the project manager is a major driver in the software equation.

Several methods of schedule and cost estimation have been available (academic and commercial) and proven since the early 1980s. These estimating methods generally consider the impact of size and the development environment on the resulting delivery schedule and resource requirements. The methods do not arrive at the resource estimates automatically. The estimator must understand the method to input correct parameters to the tool. This knowledge is only available through training and experience.

The estimating methods can also produce incorrect or misleading estimates. This project is an ideal example of estimate misuse. The contractor's proposal estimate grossly erred in the size of the development task by overestimating the availability and benefits of reusable software components. Other key issues (other than size) were ignored in proposal estimate. These omissions included the avionics application experience and JOVIAL language experience of the remote development team. Volatility of the development environment and experience with that environment at both development sites were ignored. Communication difficulties between the sites were dismissed.

The proposal cost evaluation team noticed a large discrepancy in the proposed software cost when comparing the cost proposed by the incumbent developer and the new contractor. The cost evaluation team notified the new contractor that the team believed the contractor either did not understand the tasks or that for some other reason had not bid enough engineering hours. The contractor responded that the costs had been verified using *a proven cost model* and they did not believe they made a mistake. The contractor subsequently reduced its bid about 15 percent. An experienced software estimator would have raised a serious cost risk concern following the contractor response.

Lesson 3: Estimating skill and experience is essential in software acquisition and development.

Modern Development Practice

There has been considerable effort in establishing the importance of good software practices and a manageable development process in successful software development. The trail to modern software development begins in the 1950s (before software was born) with the work of W. E. Deming[1]. Deming's work became a basis for the current Capability Maturity Model®. We all recognize that large-scale software development must be well managed to have any possibility of success. In the mid-1980s, the Waterfall Model represented the most commonly used software development approach. The impact of process and process management was yet to be defined outside of the software esti-

mating methods.

One issue that arises almost constantly is the cost and schedule impact of change. A change can be as simple as changing word processors, or as complex as changing the entire development process. How long does it take to become proficient in the Ada programming language? Thirty days? It is not likely. Historic data places Ada mastery at more than a few years. How long does it take to install a new computer network? A weekend? We have a tendency as humans to trivialize the effort to master any new technology. The larger the number of concurrent changes or magnitude of a single change, the more amount of time and cost it takes to accomplish that change. This project demonstrated the human frailty.

The contractor proposed integrating in-house tools on a state-of-the-art computing system, and supplementing those tools with government-furnished equipment software to complete the development system. The proposal also stated the need to link a remote test subcontractor to the new development system. The computer program development plan (CPDP) was still in outline form at contract award. The new technology and lack of experience present in this development environment should have triggered several risk issues. Each of the issues involved personnel training and experience, system refinement, and testing. None of the environment problems were considered in the development cost.

The contractor added a new geographically remote development site for the avionics software development almost immediately after the contract award. This new organization was not mentioned in the program proposal or in the preliminary CPDP. The new remote staff was unfamiliar with the contractor organization and development process (the CPDP had not been approved), the application area (avionics), the required programming language (JOVIAL), the development tools and environment, or the network connecting the two development sites. The contractor had not successfully ported the avionics software tools to the development computer at the time of this acquisition. No cost or schedule impact was included for this set of circumstances. All major cost estimating methods available at the time assumed a major impact.

Lesson 4: Instant experience is a myth.

Software development is largely a communication problem. Paper and electronic interfaces between software engineers have not proven to be as effective as face-to-face communication. This is primarily due to interface complexity and the development product clarity. The major software estimating tools reduce the software organization's productivity for the use of multiple organizations and/or multiple development sites.

The new software development organization that was acquired at the outset of development was not only new, but was separated by thousands of miles from the contractor's primary site. The communication between these two sites was intended to be electronic – yet another new technology that was not proven. The new personnel were not only unfamiliar with the development environment, but also had no experience or knowledge of the contractor. The organization's development practices and procedures were at best documented, however, seldom followed. The computing network between the two sites was still not operational almost two years after the contract award.

Since the software estimate totally ignored these issues, as well as the experience issues, the logical assessment is they assumed the volatility of the development environment was also no problem.

Lesson 5: Multiple development sites and organizations increase risk and decrease productivity.

Summary

Software project failure has been with us for a long time. Volumes have been written about the list of potential problem areas in the acquisition of large, complex software systems. The list has not changed much in the last 30 years. Unreal cost and schedule estimates are causes for project failure as often as inadequate technology is.

The acquisition team and the contractor must share the responsibility of this classic failure. The proposal never should have been submitted, and the contract never should have been awarded. The contractor's proposal could not have been based on their experience in the development of this type of system. Buzzwords and hype often cloud judgement, but reuse has been around much longer than the hype. Great expectations overrode common sense in their cost proposal planning and estimates.

On the other hand, the proposal cost evaluation team did not have a baseline with which the proposed costs could have been compared. The team did compare the two proposed estimates, noted the large discrepancy, and acted accordingly. The cost evaluation team notified the new contractor that the team believed the contractor either did not understand the tasks or that for some other reason had not bid enough engineering hours. The contractor response was that the costs had been verified using a proven cost model, and they did not believe they made a mistake. The contractor subsequently reduced its bid about 15 percent. The response should have at least initiated a serious analysis of the proposal.

Lesson 6: The major lesson learned from this software acquisition is "never make an uninformed decision."◆

Note

1. Dr. W. Edwards Deming is known as the father of the Japanese post-war industrial revival and was regarded by many as the leading quality guru in the United States. He passed on in 1993.

About the Author

Randall W. Jensen, Ph.D., is a consultant for the Software Technology Support Center, Hill Air Force Base, with more than 40 years of practical experience as a computer professional in hardware and software development. He developed the model that underlies the Sage and the GAI SEER-SEM software cost and schedule estimating systems. Jensen received the International Society of Parametric Analysts Freiman Award for Outstanding Contributions to Parametric Estimating in 1984. He has published several computer-related texts, including "Software Engineering," and numerous software and hardware analysis papers. He is currently preparing "Extreme Software Estimating" for Prentice Hall, Inc. Jensen has a bachelor's of science degree in electrical engineering, a master's of science degree in electrical engineering, and a doctorate in electrical engineering from Utah State University.

Software Technology Support Center
6022 Fir Ave.
Bldg. 1238
Hill AFB, UT 84056-5820
Phone: (801) 775-5742
Fax: (801) 777-8069
E-mail: randall.jensen@hill.af.mil

Defect Management: A Study in Contradictions

Raymond Grossman
L-3 Communication Systems-East

In today's defense business environment, many software and system development contracts require mandatory compliance with standardized software or integrated process models such as ISO 9000, the Capability Maturity Model® (CMM®) or CMM Integration℠. One component that all these models have in common is the use of defects as a measure of process and product quality. Entire key process areas[1] and process areas[2] are devoted to defect prevention and defect causal analysis and resolution. With such a reliance on this metric as a key indicator, it would seem that the definition and interpretation of the term defect would be universally understood and accepted. However, nothing could be further from the truth.

Just as *beauty is in the eye of the beholder*, what constitutes a defect is guided by the personal biases and the organizational position of the person making the evaluation. Is a bug found by a developer during unit testing a reportable defect, or just a *normal*, non-quantifiable part of the development process?

To some (many) developers, the latter is true. Their contention may be that the whole point of unit testing is to uncover problems; if they were forced to document every bug as a defect, the job would never be accomplished on time and on budget. However, if a process engineer were asked the same question, he or she might suggest that the uncovering of a problem during testing is an opportunity to examine the cause of the problem, to determine in what phase the defect was introduced, and to reveal what earlier process failed to find the defect.

Management, in this scenario, is at cross-purposes. On one hand it has schedule obligations and does not want to burden its developers with doing *extra* paperwork. On the other hand, a *process savvy* management is aware of the economy of finding defects as early in the development life cycle as possible and thinks any opportunity for process improvement should be pursued.

A seemingly more straightforward scenario is to document defects at peer review meetings. These types of reviews can take many forms, but their basic task is to have a product (requirements, design, code, document, etc.) reviewed by a group of interested and knowledgeable individuals. The output of peer review meetings is a list of action items or defects that must be addressed or fixed in a timely fashion. Even in organizations that have defined peer review processes, there are often areas of contention when recording defects. Many times the line that divides the definition of action items versus defects is vague, and defects are incorrectly classified as action items. In other cases, several discovered defects, which seem similar in nature, are grouped under one defect identifier. While this seems expedient, valuable information required for defect causal analysis is being lost.

Defect Management Impediments

You might wonder why it is so difficult to get buy-in from most developers and many managers if all the software process models are treating *defects* as useful and even desirable byproducts of development. The problem lies in the definition of the term defect that is found in most reference sources. As a prime example, Webster's Dictionary[3] defines a defect as "a blemish; fault, flaw." It is difficult to see the positive value in such a definition. Can you imagine a teacher of a poorly performing class meeting with the principal and being congratulated for finding all the defects in their student's test papers? Or can you imagine the president of an automobile company who gives bonuses to his engineering staff after they have failed several crash-test categories? We have been programmed to equate defects as failures against the developer, teacher, manufacturer, etc. Is it any wonder that a developer would be reluctant to *admit* finding defects in his or her work product?

> *"We have been programmed to equate defects as failures against the developer, teacher, manufacturer, etc. Is it any wonder that a developer would be reluctant to admit finding defects in his or her work product?"*

Even in the best-case scenario in which developers record defects during peer reviews and testing phases, there are practices employed in the development life cycles that make the use of defects, as a quality measure, questionable. Among these is the correcting of problems and errors prior to the formal peer review processes. Commonly during all life-cycle phases, regular discussions occur between developers and system and test engineers. In many cases, these discussions lead to changes in documentation, design, and/or code. These changes are, in the majority of cases, not formally documented. By the time formal peer reviews for these products are conducted, most of the questionable details have been resolved leading to few if any defects uncovered at the reviews.

From a process quantitative perspective, little can be learned from these reviews. All the pre-review defect information has been lost due to the lack of documentation; defects found at the reviews, which potentially could be used to provide insight into the capability of the review process, are minimal. The statistical process charts tracking these reviews over time may resemble a *flat-line* electrocardiogram with little or no defect variation. Figure 1 shows the results of reviews that have been pre-reviewed without defects being documented. Figure 2 shows a *normal* distribution of defects over time (note: points are almost equally above and below control limit midline).

Testing processes present their own

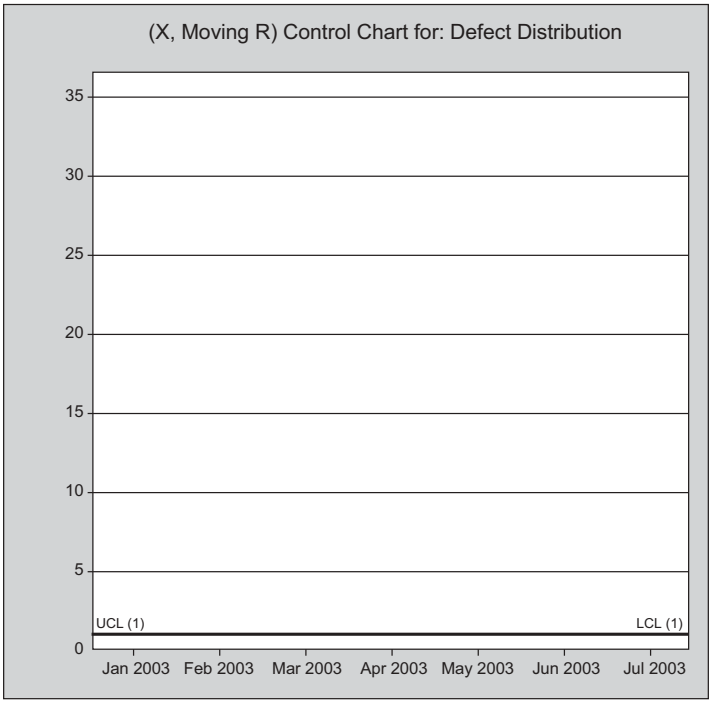


Figure 1: *Distribution of Review Defects with Pre-Review Defects not Documented*

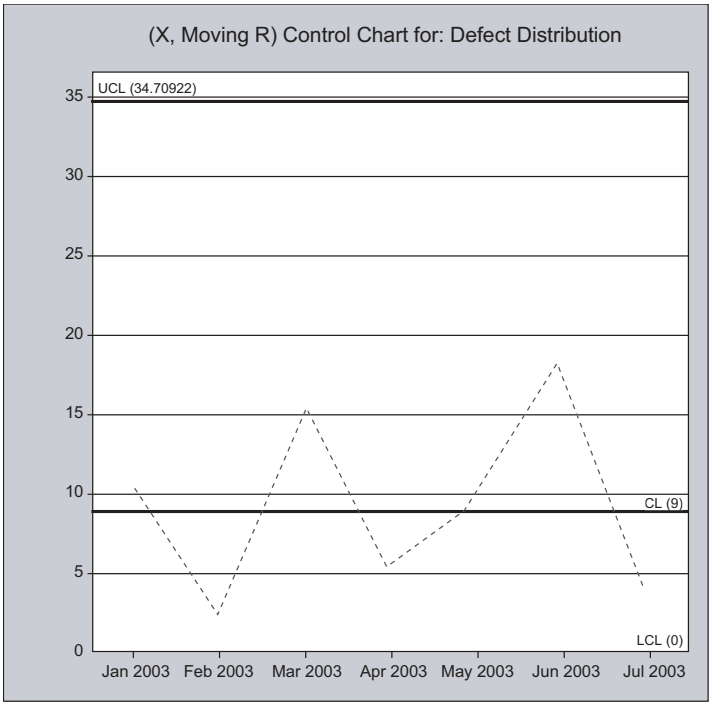


Figure 2: *Normal Distribution of Review Defects*

unique impediments to accurate defect reporting. In the traditional *Waterfall Model* for software development, *unit testing* is the process immediately succeeding *coding*. It has the potential of removing a great deal of defects and uncovering weaknesses in earlier life-cycle processes. Yet, in many organizations, few or no defect measurements are recorded and analyzed. Usually, the only quantitative measures that can be determined are the number of passed and failed unit tests.

Even in organizations that do capture unit test defects, the documentation is vague and usually does not include casual analysis parameters that would pinpoint the life-cycle event that failed to detect the defect that was found during the unit test. Once again the prime culprit for the lack of defect reporting is fear of retribution by the developer or team leader. Unlike other tests such as system-level testing, the individual who created the code normally performs unit testing. To report defects that are found during unit testing is, in their minds at least, an admission of incompetence.

A second but no less common explanation for the lack of defect reporting is *time*. If done diligently, it takes some effort to analyze each defect found, to determine why it was introduced, and to specify which prior process failed to find it.

While both of these explanations are understandable, it still makes the unit test process far less useful as a quality tool. The same problems may be found in the software integration phase in those companies where the developer of the code also writes the integration test procedures and performs the tests. In some organizations, however, independent test groups perform this phase so defect reporting may be more accurate and complete.

While the usefulness of defect management analysis is compromised by the sins of omission, as in the non-reporting of defects illustrated in the previous cases, equally damaging is the reporting of inaccurate defect information. Bad defect reporting leads to the phenomenon sometimes known in the computing world as *garbage in, garbage out*. Simply put, if the data reported is bad, the analysis and resulting conclusions will be wrong as well.

The reasons behind the incorrect reporting of defects may range from expedient reporting of defects on mandatory forms to the misunderstanding of defect definitions (as discussed earlier in this article). In the process-oriented environment found in most defense organizations today, developers are required by their organization's standard defined processes to transcribe defects at various life-cycle milestones. Pressured by schedules and wary of possible management criticism, some developers may choose to fill in the required defect fields with information that does not necessarily reflect the true nature of the problem and/or cause. The resultant analysis may lead to incorrect conclusions about the causes of the defects and thus dilute or eradicate any benefit derived from the process.

Utilizing Defects for Quality and Process Improvement

Up until this point, this article has painted a rather bleak picture of the utility of defects as a quality measure. Typically, metrics such as *peer review defect density* will yield little benefit as a measurement of quality if only small numbers of defects are being reported during the process. However, there are still process improvement opportunities to be gained by determining the root causes[4] of those defects that are reported during the software development life cycles.

If a particular phase seems more prone to producing defects, the related process may have to be reviewed and subsequently updated or changed. This idea can be applied to the examples presented earlier in order to show the possible benefits of finding and analyzing defects. If the school teacher with the poorly performing class uses those negative test results as a basis for changing his or her style or teaching methods to be more in compliance with the students' special needs, the performance of the students may improve. Similarly, if the poor crash-test results in the automobile company cause the improvement of the related quality assurance methods, future test results will most likely be positively affected.

Thus far, only defects found during the software development life-cycle processes have been considered. Once a product has been put under configuration management and delivered to the cus-

tomer (internal or external company), defects are uncovered during field-testing, system testing, and normal customer usage. These defects are then reviewed, verified, and categorized and submitted back to the development team for resolution. These defects differ in several ways from those discovered during the development life cycle.

First, the reporting reluctance the developer wrestled with during the development life cycle is eliminated. The defect has been found and now must be dealt with. In the same vein, the *time-to-document-the-defect* dilemma has been removed from the developer's plate. Time and budget have been allotted to analyze and fix the defect. Finally, the fear of retribution is, if not eliminated, made less significant. The defect exists and, no matter how it was caused or whose *fault* it was it now must be corrected.

The key benefits of post-development cycle defects are that they can be used as the basis of product and process quality measurements as well as for process improvement identifiers. Quality metrics, such as product defect density, can be calculated from the *post-development cycle* defects and used as a basis of comparison against other projects in order to set product goals and to predict future budgets and schedules. Performing causal analysis on post-development cycle defects will enable the process team to determine the development cycle processes' defect insertion, removal, and leakage rates and make the appropriate process adjustments.

The Team Approach to Improving Defect Management

In an effort to assist developers in distinguishing true defects from action items and to expedite the proper use of techniques such as root-cause analysis and quantitative process management, some organizations have established defect analysis teams (DATs). These teams, usually consisting of members of the organization's Software Engineering Process Group, are tasked with reviewing the defect documentation generated by developers during the various development processes.

DATs examine questionable items such as missing fields, inconsistent or contradictory defect descriptions, or flag other anomalies. Interviews with the appropriate developers are held where the team's findings are discussed. At these meetings, the developer and the team review defect definitions and related causal analysis concepts, and the defect documentation is completed to all participants' satisfaction. At times, even presumably correctly completed documents are discussed with developers in an effort to reinforce the defect management concepts.

In most organizations employing this teaming technique, it is the goal of the DATs to review selected documentation from every appropriate software project in the organization. The employment of DATs helps foster more consistent and complete defect reporting and may act as a catalyst to the institutionalization of defect management techniques throughout the organization.

"The reasons behind the incorrect reporting of defects may range from expedient reporting of defects on mandatory forms to the misunderstanding of defect definitions."

Conclusion

In the best of all possible worlds where no schedules, egos, or calendars exist, defect management would be an extremely useful and informative process in all phases of software development and maintenance. However, due to the reasons mentioned in this article, the utility of defects is diminished during the software development processes. Is this an insurmountable problem with no corrective actions possible? The answer largely depends on factors outside the control of the software developer.

Training must be given and constantly reinforced to change the perception of defect reporting being purely negative and self-indicting. The value of uncovering defects and concepts of causal analysis must be part of this training. Management must also be made aware of the cost and maintenance savings opportunities in defect removal in the earliest possible life-cycle phase. The employment of DATs is an effective way of mentoring and reinforcing defect reporting concepts and processes. Finally, but most importantly, management must assure developers that it supports and encourages defect identification and documentation.

While these suggestions are few, they may look daunting to organizations who have long histories of treating defects purely as problems and to developers who believe that their performance reviews are negatively affected by the number of defects found in their products.◆

Notes

1. This is Capability Maturity Model® nomenclature.
2. This is Capability Maturity Model Integration nomenclature.
3. PSI & Associates Inc. New Webster's Dictionary & Thesaurus. 1991.
4. The root cause is the identified reason for the presence of a defect or problem. The most basic reason, which if eliminated, would prevent recurrence. <www.isixsigma.com/dictionary/Root_Cause>.

About the Author

Raymond Grossman is a senior software process engineer with L-3 Communications-East in Camden, N.J. He is responsible for creating and implementing a metrics program and in helping the organization transition to the Capability Maturity Model® Integration model. Grossman has more than 30 years experience in all phases of software development, including project management. For the past seven years, he has been involved in process improvement and software measurements and metrics, and has been a key player in several Capability Maturity Model (CMM) evaluations. Grossman was part of a team that helped his former organization achieve a CMM Level 5 rating. He has a bachelor's degree in engineering from the City College of New York and has been awarded Certified Professional Management Credentials from James Madison University.

L-3 Communication Systems–East
1 Federal St.
Camden, NJ 08103
Phone: (856) 338-6220
Fax: (856) 338-2425
E-mail: raymond.grossman@L-3com.com

Defect Mismanagement

Back in 1974, I was an application programmer at the U.S. Air Force Strategic Air Command (now STRATCOM) headquarters, Offutt Air Force Base. I worked in support of targeting. My end-user, which at the time was a Marine lieutenant colonel, needed a simple program to filter out spurious data from digitized photoreconnaissance data. Recognizing my true worth as a top-notch applications programmer (and not having anybody else to support his needs), he asked me to write the program for him.

In gathering the requirements, I noticed that there would be hundreds of data sets each night. Obviously, I couldn't run 100 different jobs, so I grouped the data together into one large nightly run. I needed a card (yes, it was that long ago – we used punched cards) to separate the data streams, so I asked the end-user what special characters would be present in the final data. The user replied, "No special characters are used at all."

Since I couldn't get real data to test (security issues), I faked up some data separating each data stream using a single card with a '!' in column one. My test worked, and I cut the program over to the operational side of the system.

The very next morning at 2 a.m., I received a call from the computer operator (yes, it was that long ago – we had real live operators to run the card decks) telling me that my job terminated with 0.001 seconds of CPU time. Now, I am a hotshot coder, but nothing ran that fast in Cobol (yes, it was that ... oh, you know). I threw on a uniform, went in, and after getting permission to access the real data, poured through my core dump (yes, it was that ... never mind) and saw that the very first data stream had multiple '!'s embedded in it. My program had seen each '!' as a data stream separator, and did not find enough data in each stream to analyze.

Being righteously indignant, I was waiting for my user to show up at 0730. I quickly pointed out to him that he had asserted that there would be no special characters, but '!' was used frequently in the data. His response was to say, "Well, '!' isn't special. We use it all the time." At that point, I became a Zen master when I realized that the user and I did not speak the same language. What I had logged as a defect was, in reality, nothing more than a simple miscommunication. Isn't it funny how that often tends to be the case?

One of the major failings in the way we currently develop software is that so many things can hide under the cover of a defect. Forgot to include a feature? It's a defect. Have bad code? It's another defect. As a Personal Software Process℠ instructor, I teach that not all defects are created equal. Some come from simple problems and are simple to fix. Others have complex causes and are complex to fix. Frequently, we have defects resulting from simple errors that are costly to fix. And, very occasionally, we hit those that have complex causes, but are simple to fix.

In BackTalk, we sometimes write columns that are cynical and sarcastic. Sometimes we write columns that are funny. And – as in this column – sometimes I get to point out the obvious. The following are "obvious things about defects you (1) probably already know, or (2) should know, or (3) wish your user/developer knew:"

1. If you have ever thought of it as something that can go wrong, it's not unexpected any more, is it? It really can't be an error anymore – it's something you thought of, made a value judgment/risk assessment about, and decided to ignore.
2. If any error can occur, it will eventually. At 3 Gigahertz, lots of instructions are executed every second. Eventually, the most bizarre timing occurrences happen.
3. There is no such thing as foolproof. Nature and genetics are producing 248 births each minute. Some of them are bound to do foolish things at an alarming rate. Don't ever think, "Nobody would be so dumb as to ..." They are. They will. It's your fault.
4. Quit hiding poor requirements as defects. Make sure you do root-cause analysis on each defect – and fix the problem, not the symptom. If you have lots of defects due to poor requirements elicitation and validation, fix your requirements process instead of just hiring additional developers to fix the so-called defects.
5. Don't settle for large numbers of defects. If your code is bad, train and educate your developers to make better code. If your design is bad, get some true designers to help with your architectural, data, and interface design. Designing code is not what design is about. Most of your errors will occur due to poor architectural design, poorly thought-out interfaces, or inefficient data. On the other hand, if your defects are from poor requirements, see No. 4.
5. Developers, you have to give the users what you agreed to give them. If you need to slip requirements or postpone deliverables, communicate that to the users. If you surprise them with incomplete or defect-ridden code, they are going to be unhappy.
6. Users, you have to explain what you need to get code that meets your needs. If you won't commit to what you really need, then you won't get what you really need.
7. Developers, before you move to the next step in your life cycle, verify and validate what you have. There is no use proceeding until you know that what you have so far is correct.
8. Users, if you want it badly, you'll get it badly. If you ask for too much, it will take too much time. You have to compromise. Understand that it takes time to create a software product. You have a choice – the more you push for fast delivery, the less quality you will get.
9. No matter how bad the defect is, it could have been worse. In fact, wait a while, and you'll remember your current problem as the good old days. (OK, I just couldn't keep the cynicism and sarcasm totally out).

– **David A. Cook**
david.cook@hill.af.mil
Software Technology Support Center/
Shim Enterprises Inc.

P.S. It's a column about defects – of course the list is misnumbered!

TOP 5 QUALITY SOFTWARE PROJECTS

2003 U.S. GOVERNMENT'S

Top 5

IF YOU WOULD LIKE INFORMATION OR TO ENTER, PLEASE VISIT OUR WEB SITE

www.stsc.hill.af.mil/crosstalk

Sponsored by the Computer Resources Support Improvement Program (CRSIP)

Published by the Software Technology Support Center (STSC)

CrossTalk / MASE
6022 Fir Ave.
Bldg. 1238
Hill AFB, UT 84056-5820

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