Nuts Bolts Nuts Bolts

By culling systematic errors from problem reports, DCA helps develop strategies for preventing defects or detecting them earlier.

Learning from Our Mistakes with Defect Causal Analysis

David N. Card, Software Productivity Consortium

Defect causal analysis offers a simple, low-cost method for systematically improving the quality of software produced by a team, project, or organization. DCA takes advantage of one of the most widely available types of quality information—the software problem report. This information drives a team-based technique for defect causal analysis. The analysis leads to process changes that help prevent defects and ensure their early detection.

Although some approaches to quality improvement involve exhaustive defect classification schemes or complex mathematical models, the approach I present relies on basic techniques that can be implemented readily by the typical software organization. The DCA process was developed at IBM[1]; I adapted it for Computer Sciences Corporation[2] and other organizations. Three key principles drive the DCA approach.

♦ *Reduce defects to improve quality.* Although there are many different ideas about what quality is or which "-ility" is more important—reliability, portability, or whatever—we can all probably agree that a product with many defects lacks quality, however you define it. Thus, we can improve software quality by focusing on the prevention and early detection of defects, a readily measurable software attribute.

Welcome to Nuts and Bolts, a new section of articles that will help software practitioners apply specific technologies or development tasks to their work. As an *IEEE Software* reader, you're uniquely qualified to contribute to Nuts and Bolts. Your submissions should contain methods that have proven successful and readily implementable by the typical software organization. These methods need not, however, be widely established in practice yet.

Our Tutorial articles will explain how a specific technology is applied to software development or how a specific real-world problem can be solved. It surveys all aspects of a research or application domain. Given our magazine's broad scope, contributions should be written at a level easily understood by nonexperts.

Our How-To articles are new to *IEEE Software*. They will give developers and managers step-by-step instructions for doing one thing one way—and well. Again, any given technique will certainly be completely new to some readers, or your article may provide them with the key to applying it successfully. Thus, How-To's should be short (3-5 pages) and to the point.

Detailed guidelines for writing *IEEE Software* How-To and Tutorial articles can be found at http://computer.org/software/genres.htm, or request them via e-mail to software@computer.org. If you have questions about submitting articles to Nuts and Bolts or would like to discuss potential article topics, please contact me directly at card@software.org.

—David N. Card, Nuts & Bolts editor

♦ *Apply local expertise.* The people who really understand what went wrong are the people present when the defects were inserted—the software engineering team. Although causal analysis can be performed by outside groups such as quality assurance, researchers, or a software engineering process group, those people generally don't know the detailed circumstances that led to the mistake, or how to prevent it in the future.

♦ *Focus on systematic errors.* Most projects have too many defects to even consider conducting a causal analysis of them all. However, some mistakes tend to be repeated. These "systematic errors" account for a large portion of the defects found in the typical software project. Identifying and preventing systematic errors can have a big impact on quality (in terms of defects) for a relatively small investment.

There are relatively few prerequisites for implementing DCA.

♦ You must have defects—usually not a difficult prerequisite to satisfy.

♦ Your defects must have been documented through problem reports, inspection results, and so on.

♦ You must have a desire to avoid mistakes or at least the negative feedback associated with them.

♦ You must define a basic software process to provide a framework for effective actions.

You don't have to fully define and detail your software engineering process for DCA to provide value. However, without at least a basic process framework it becomes difficult to understand where and how defects enter the software and which actions are most likely to prevent or find them earlier.

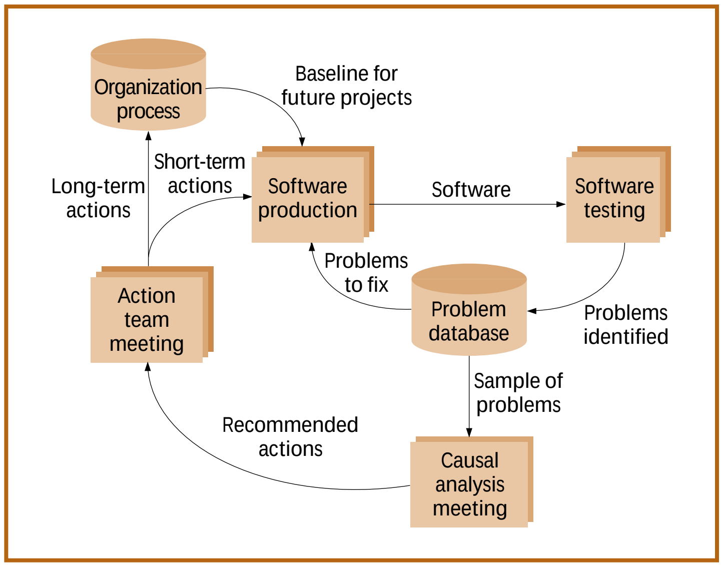


Figure 1. The defect-causal-analysis process, which draws problem samples from the database for periodic meetings that produce action team recommendations. Short-term proposals benefit the current project; long-term proposals are integrated into the organization's process improvement efforts and benefit future projects.

Process overview

Typically, software defects are fixed and forgotten. DCA provides a mechanism for *learning* from them. Figure 1 shows the DCA process. Software is produced, then undergoes testing by either the programmers or a separate test team. Problems identified during testing are recorded in a problem database. Eventually, programmers get around to fixing them. In the typical software organization, that's usually the last time anyone thinks about the problems. However, DCA adds steps that help the organization learn from its mistakes instead of continually repeating them.

The DCA process draws samples of problems from the database for periodic causal-analysis meetings. These meetings produce recommendations for an action team. The proposals may be either short-term (for immediate action by project staff) or long-term (for

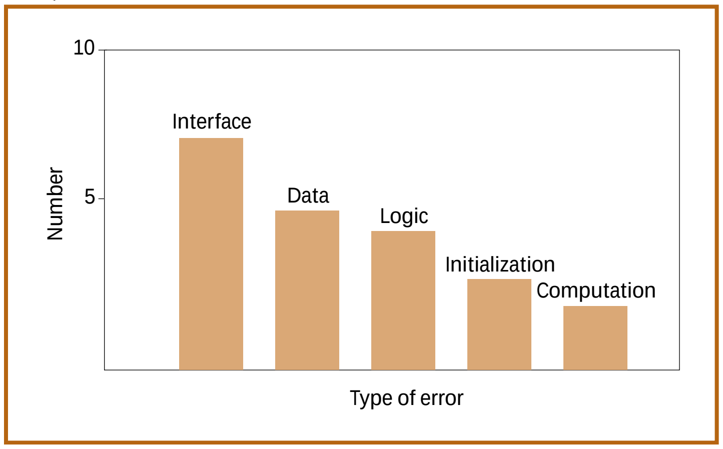


Figure 2. This Pareto chart example shows that Interface defects comprised the largest defect class.

the benefit of future projects). The longer-term proposed action should be integrated into the organization's other process improvement efforts. Usually, the effects of causal analysis and resulting actions can be measured within a few months of the project's start.

Causal-analysis team

The causal-analysis team is the focus of the DCA process. It performs most of its work during a causal-analysis meeting in which the software team analyzes problems and recommends improvements that will prevent defects or detect them earlier in the process. The causal-analysis team should consist of the software producers (developer and maintainers) who have the greatest intimacy with the product and process. Robert Mays and colleagues[1] observe that the best results are obtained when DCA is performed "by the developer making the error, [which] results in a more accurate determination of the cause of the defect and more relevant preventive actions."

Causal-analysis meetings should be held at regular intervals, typically

- after each testing phase,
- after each development phase, and
- three months after the delivery of each release.

Meetings typically last about two hours. Encourage the entire development team to participate. Usually only 50 to 60 percent of the team attends any given meeting. By limiting teams to 25 people or less, you get meetings of a manageable size. If the project is very large, define subteams logically. For example, you can establish subteams for the database, applications, and systems programmers. Each subteam should investigate problems from its own domain.

The causal-analysis meeting should be led by a designated moderator or facilitator. The role of the facilitator is to hold the team to its agenda while preserving group integrity. The facilitator could be a respected member of the team, a member of the software engineering process group, a tester, or a member of another causal-analysis team. Don't select a manager or quality assurance person as the facilitator because that individual may inhibit discussion. As we shall see, management and quality assurance have other important roles to play in the DCA process.

The typical agenda for a DCA meeting includes the following steps:

- select problem sample,
- classify selected problems,
- identify systematic errors,
- determine principal cause,
- develop action proposals, and
- document meeting results.

The output of each step provides input to the next. The final results of the causal-analysis meeting are specific recommendations provided to the action team, usually the software engineering process group.

Select problem sample

Because most projects have more problems than they can afford to analyze, a sample must be selected for consideration during the causal-analysis meeting. Do not select more than 20 problems for analysis in one session. The moderator or the custodian of the problem report database (often QA) may perform this step in advance of the group meeting.

Make the sample as representative of the team's work as possible. Omit obvious duplicates and non-software problems. Do not restrict your sample to high-priority problems. Priority is accidental. For example, a misspelled word on a display may be cosmetic, while the same misspelling in another place may produce a system failure. Do not select problems from just one source, whether that source is an individual programmer or a system component.

Classify selected problems

Classifying or grouping problems helps to identify clusters in which systematic errors are likely to be found. You should select the classification schemes to be used when you set up the DCA process. Moreover, the meeting itself will go faster if you classify the problems to be analyzed according to a predefined classification scheme. Ideally, each problem should be classified by the programmer when implementing its fix. Alternatively, the moderator may classify the problems prior to the group meeting. Three dimensions are especially useful for classifying problems:

- When was the defect that caused the problem inserted into the software?
- When was the problem detected?
- What type of mistake was made or defect introduced?

The first two classification dimensions correspond to activities or phases in the software process. The last

Table 1
Examples of Causal-analysis Results

Domain	Problem	Cause	Solution
Scientific data processing	Members of the software development team had configured their workspaces differently with different directory structures, file protections, default devices, and so on. Each team member could get their assigned code to function in their workspace, but once it moved to integration and test it often failed to work.	Inconsistent use of development environment.	Create a standard workspace based on the operational environment in which each developer checked out their code before delivering it to integration and test.
Spacecraft navigation	Many problem reports involved consistency between different parts of the user interface, as well as discrepancies between what the user interface provided and what the software applications expected. The user interface was a purchased package, but not well documented.	Inconsistent use of user interface package.	Define standard conventions for the user interface such as position and width of fields, data editing applied to fields, and use of function keys.
Computer-aided design	The software process involved a significant delay (due to approval and scheduling) from the time that requirements were documented to the time that development actually began. Even though the same people worked on both the requirements and development, significant mistakes occurred simply because people forgot some of the decisions that had been made earlier.	Insufficient familiarity with requirements.	Conduct a requirements review immediately prior to beginning the development of each software segment.

dimension reflects the nature of the work performed and the opportunities for making mistakes. Some commonly used error types include interface, computational, logical, initialization, and data structure. Depending on the project's nature, you can add other classes such as documentation and user interface.

You can produce tables or Pareto charts to help identify problem clusters. A Pareto chart is a bar chart that shows the count of problems by type in decreasing order of frequency. Figure 2 shows a Pareto chart of defects by type for a scientific data processing system. The figure shows that Interface defects comprised the largest class. The set of problem reports comprising this class would be a good place to start looking for a systematic error.

Identify systematic errors

A systematic error results in the same or similar defects being repeated on different occasions. Usually systematic errors are associated with a specific activity or part of the product. Ignore the effects of individual defects in *seeking* out systematic errors. Table 1 shows three examples of systematic errors from different domains. In each of these cases, 20 to 40 percent of the problem reports in the samples examined during the causal-analysis meeting were associated with the common problem (systematic error) identified in the table. That many defects result from these situations is good motivation for changing the process. All the solutions described in the table required additional developer effort in the short term. Nevertheless, the developers themselves proposed these actions. Consequently, once management authorized them, these process changes were readily adopted by the engineering teams.

Determine principal cause

Many factors may contribute to a systematic error. Usually it is uneconomical to address them all, so you must concentrate attention on the principal cause. Often the principal cause will be obvious from the problem statement. Table 1 lists some examples of causes. Look for unexpected circumstances that repeat and for departures from normal practices. If the principal cause isn't obvious, develop a cause–effect diagram to try to draw it out, as shown in the boxed text on page 60. Causes usually fall into one of four categories[3]:

- ♦ methods, which may be incomplete, ambiguous, wrong, or unenforced;
- ♦ tools and environment, which may be clumsy, unreliable, or defective;
- ♦ people, who may lack adequate training or understanding; and
- ♦ input and requirements, which may be incomplete, ambiguous, or defective.

The causal categories help group related items, as well as identify general software process areas that may need attention. As with the defect classification scheme, you should adapt the causal categories as necessary to support the analysis and reporting needs of the organization.

Develop action proposals

Once you find the principal cause of a systematic defect, you must develop action proposals that will promote either prevention or earlier detection of the

Cause-Effect Diagram

The cause-effect diagram, also known as an Ishikawa or Fishbone diagram, is a simple graphical technique for sorting and relating factors that contribute to a given situation. A team usually develops the cause-effect diagram in a facilitated brainstorming session.

Figure A shows the three steps for constructing a cause-effect diagram.

1. *State the problem being analyzed.* Use the statement of the systematic error. Draw the main branch of the diagram connected to the problem statement. Insert headings for the generic causes to be used in the diagram; methods, tools, people, and input are commonly used.

2. *Brainstorm to collect specific causes.* Attach each specific cause to the appropriate generic cause or to a related previously stated cause. Several layers of causes may be attached.

3. *Once all ideas have been collected, discuss the significance of the causes identified.* Highlight the principal or operative cause contributing to the systematic error. Often this is found in the densest cluster of causes on the diagram.

The cause-effect diagram provides a focus for discussion. It does not lead to a deterministic solution. Teams working with the same information usually produce similar, but not identical, results.

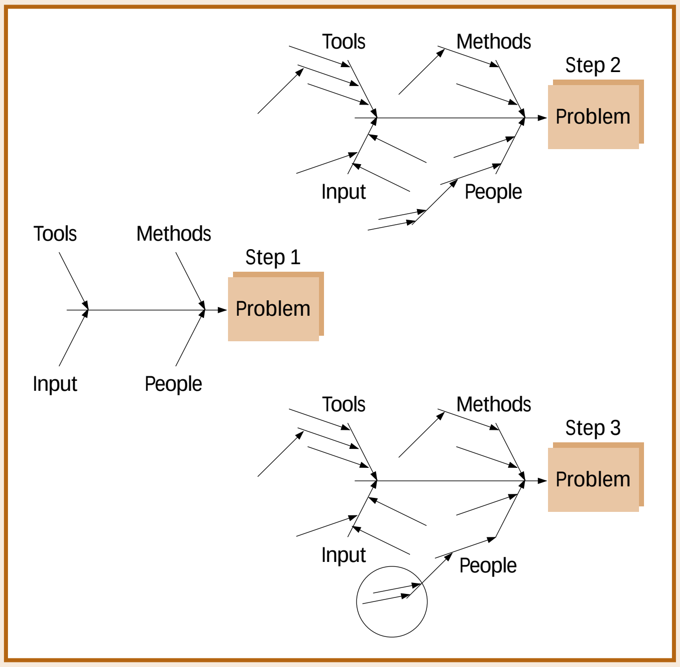


Figure A. The three steps for constructing a cause-effect diagram: (1) State the problem, (2) brainstorm to collect specific causes, and (3) discuss the significance of the causes identified.

systematic defect. Ask questions like the following when trying to elicit recommendations.

- How could we have done things differently?
- What did we need to know to avoid this?
- When is the earliest this defect could have been recognized?
- What symptoms suggest the defect will occur?

Limit the number of proposed actions and focus on those recommendations most likely to have a significant impact. Some organizations seem to measure the success of their DCA process by the number of actions proposed. Too many actions can overwhelm management's ability to process them. One good action proposal implemented is worth more than any number waiting in queue. Table 1 lists some examples of actions. Other examples include

- adding the identified defect to the list of common defects used in design inspections,
- providing training to programmers in the use of the configuration management tool, and
- notifying programmers and testers via e-mail when interface specifications change.

Make action proposals specific, as shown in these examples, and avoid general terms such as "better," "more," "as needed," "available," and "enough." State who should be responsible, what should happen, and when. DCA differs from generic process assessment methods in that it focuses on identifying specific actions rather than suggesting broad areas for increased process improvement attention.

Action proposals must focus on the systematic error. The defects associated with this error motivate taking action. Extraneous suggestions only reduce the impact of the team's findings. Avoid trying to fix someone else's process. Confine recommendations that extend beyond the scope of the team's responsibility to the interfaces between processes.

Document meeting results

You need records of meeting results to ensure that actions get implemented. In most cases a simple form-based report will suffice. Include the following information:

- identification of meeting event, such as team, date, and so on;
- description of systematic errors;
- principal cause and category of cause for each systematic error;
- list of problem reports or defects related to each systematic error; and
- proposed actions.

The problem reports, defects from inspections, and so on associated with the systematic error provide the justification for the proposed action, so it is important to enumerate them when documenting meeting results.

Action team

Most recommendations produced by the causal-analysis team can't be implemented without management support. To get any benefit from DCA, you must form an action team that meets regularly to consider proposed actions. Multiple causal-analysis teams may feed into one action team. The action team performs the following duties.

- Prioritizing action proposals. Few organizations can afford to implement all proposals at once.
- Resolving conflicts and combining related proposals, especially if multiple causal-analysis teams are operating.

- ♦ Planning and scheduling the implementation of proposals.
- ♦ Allocating resources and assigning responsibility for proposal implementation.
- ♦ Monitoring the progress of implementation and the effectiveness of actions.
- ♦ Communicating actions and status to the causal-analysis teams.

An existing organizational element, such as a software engineering process group, may serve as the action team. However, this group must have management participation or a management sponsor who can initiate action. The benefits of DCA are lost without timely action. Therefore, you should identify the members of the action team before conducting any causal-analysis meetings.

Implementing DCA

Despite DCA's simplicity, appropriate preparation increases the probability of a successful implementation. Implementing a DCA process involves three major steps.

Step 1: Define the DCA process

Before the process can be taught or executed it must be defined. Some important decisions must be made during this step.

- ♦ When will causal analysis be conducted?
- ♦ How will the DCA teams be organized if more than one is needed?
- ♦ Who will sit on the action team?

In addition to the DCA procedure, the defect classification scheme and reporting form must be documented. Data collection mechanisms must be established or augmented to support the evaluation of the DCA process.

Step 2: Provide training to participants

Everyone is likely to perform better if they understand their roles in the DCA process. I recommend three types of training.

- ♦ *Moderator training.* The causal-analysis team will run more smoothly if facilitated by someone who understands the basic techniques for promoting participation and building consensus while preserving group integrity.
- ♦ *DCA team training.* Causal-analysis teams must understand the DCA process, as well as the basic tools used in arriving at principal causes.
- ♦ *Management orientation.* Although managers don't have to understand the details of the causal-analysis meeting and techniques, they must grasp DCA's principles if they are to take action effectively.

In addition to the formal training described earlier, observing and providing feedback on a team's first meeting has proven effective in assessing their understanding of the DCA process and correcting any misinterpretations or misunderstandings.

Step 3: Evaluate the DCA process

The DCA process should evolve over time to better suit the organization's needs. Two sources of information should drive this evolution:

- ♦ Participant feedback on the DCA process itself. This includes the procedure, reporting forms, training, and classification scheme. This can be accomplished via a simple survey of team members.
- ♦ Quantitative data on the effects of DCA-originated actions. Ideally, an organization should establish a baseline of defect rates and types prior to commencing DCA. Measuring rates and types after actions have been taken can show the effects of DCA.

Potential problems

Because DCA is a commonsense idea, organizations sometimes attempt to "just do it" without adequate preparation. Planning, training, and follow-up help avoid some common pitfalls of DCA implementations.

Despite DCA's simplicity, preparation increases the odds of successful implementation.

Other dangers can result from the fear of blame that Deming recognized years ago. Behaviors to avoid include the following.

- ♦ *Fixing someone else's process.* It is always tempting to try to fix the customer's or tester's process rather than our own. This isn't an effective use of time because producers have neither the expertise to suggest effective changes to the other processes, nor the control to implement these changes.
- ♦ *Defending our process.* No one likes to make mistakes. DCA can become a forum for rationalizing away errors—downgrading their severity and converting them from "problems" to "changes." Consequently, the "numbers" may look better, but performance won't improve.
- ♦ *Discounting the producers.* Since many software managers previously worked as software engineers, they have a tendency to think that they know better than the people actually performing the work. Modifying the producers' proposals to conform with management's preconceptions reduces the enthusiasm of producers for implementing the resulting actions.

Other potential problems relate to the management of the DCA process itself, including the following.

- ♦ *Adopting a suggestion focus.* The goal of DCA is to motivate action. Nevertheless, organizations often are tempted to treat DCA like suggestion programs in which score is kept by counting the proposed suggestions or recommendations rather than focusing on actions.
- ♦ *Procrastinating.* It takes time to realize the benefits of any improvement program. However, a significant part of this lag often can be attributed to management's delay in implementing recommendations. Resources for action should be set aside before initi-

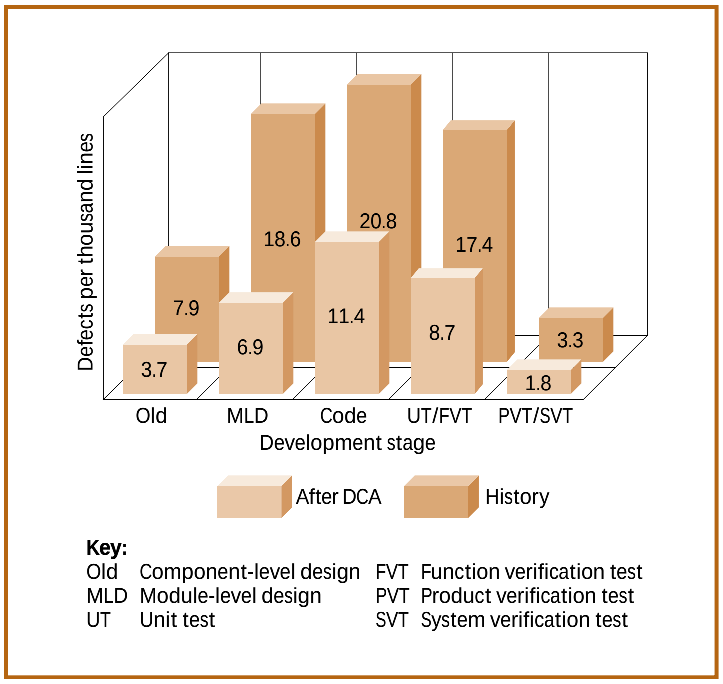


Figure 3. IBM's experience with DCA showed that defect rates declined by an average of 50 percent. Computer Sciences Corporation enjoyed similar benefits when it implemented DCA.

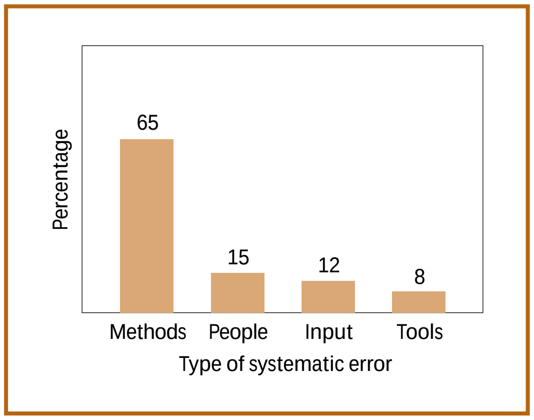


Figure 4. Distribution of systematic errors for one organization. About two-thirds of all systematic errors stemmed from flawed software methods, especially failures of process or communication.

ating the DCA process.

♦ *Letting the process drift.* Both the analysis and action elements of DCA are human-intensive processes. Unless you actively manage them, social forces can change their methods and goals, leading to the problems described previously.

Enthusiastic follow-through by management and respect for the technical skills of the software producers are the keys to a successful DCA implementation.

DCA'S impact

DCA's impact on an organization can be assessed from at least three perspectives: its benefits in terms of quality improvement, the cost to implement DCA, and its compatibility with other improvement initiatives.

Evidence of improvement

The two simplest ways of determining the effectiveness of improvement actions on quality are to track the overall defect rate and the distribution of error types. Here, I summarize the benefits of DCA from the perspectives of two different organizations. The IBM experience involved projects with hundreds of staff,[1] while, in contrast, the Computer Sciences Corporation experience involved projects with only tens of staff.[2] In both cases, defect rates declined by about 50 percent over each two-year period of study. Further, CSC experienced the lowest defect rate achieved in the application domain studied and "severe" errors disappeared[2] from the project that implemented DCA. This project received a commendation from the customer for the quality of the software delivered.

Figure 3 shows that in IBM the defect rate declined in all phases of the life cycle compared to a historical baseline of projects from the same application domain.

Not all defects equally affect the user and the product's success. For example, certain parts are critical to system operation; others are not. Nevertheless, reducing defects in general reduces critical defects, too—as the CSC example shows. Moreover, reducing systematic sources of defects usually reduces development cost and cycle time. Both IBM and CSC enjoyed significant dollar savings as a result of systematic quality improvement. Focusing exclusively on the egregious mistakes that crash the system usually doesn't produce these side benefits.

The distribution of error types may be a more sensitive indicator of process changes in the short-term.[4] For example, if a systematic error had been found among the Interface problems in Figure 2, and an effective action for preventing this systematic error had been implemented, the proportion of problems in this class should diminish over time. This change in distribution often can be detected earlier than the change in defect rate.

Cost of implementation

DCA isn't expensive, but it does require some investment. The cost of operating a DCA program, from causal analysis through implementing actions, ranges from 0.5 percent of the software budget at IBM[1] to 1.5 percent at CSC.[2] Beyond this operational cost, a startup investment must be made to fund training, classification scheme definitions, procedures setup, and the establishment of data collection mechanisms. This implementation cost depends on what relevant resources the organization already possesses as well as how widely DCA is to be implemented. Nevertheless, the experiences I've cited show that if quality is important to your organization, DCA is well worth the investment.

Relationship to CMM

The Software Engineering Institute's Capability Maturity Model envisions the deployment of DCA and other defect-prevention techniques at Level 5, Optimizing.[5] Consequently, many organizations put off investment in DCA because they want to focus all their attention on exactly the level they are trying to attain currently—and that usually isn't Level 5. However, the CMM is a descriptive, not prescriptive model: it describes what must be in place for an organization to qualify for each level, not *how* an organization moves from one level to another. Moreover, some practices exert a maturity-pull effect.[6] Often, the most effective way of moving from one level to the next is to adopt some practices from higher levels. For example, an organization moving from Level 1 to Level 2 usually starts by establishing a training program and software engineering process group—Level 3 practices!

DCA is a maturity-pull technology for organizations attempting to establish themselves at Levels 3 and 4 because it facilitates the following behaviors.

♦ *Quality awareness.* Participating in causal-analysis and action meetings makes software quality tangible to managers and producers alike. They gain a practical understanding of the consequences of quality.

♦ *Process commitment.* Evidence accumulated through DCA helps show producers the value of conforming to the process. Figure 4 shows data on one organization's systematic error distribution. About two-thirds of all systematic defects stemmed from flawed software methods. Moreover, most problems from this class were caused by a failure to follow some element of the defined process or to communicate important information. Conducting DCA convinces software producers of the value of an effective process.

♦ *Quality measurement understanding.* When software producers begin analyzing problems and implementing improvements via DCA, they begin to understand the value of quality data. Their fears about data misuse decline, because now they are data users too.

Many organizations have adopted strategies other than the CMM to drive process improvement. DCA provides an effective mechanism for feedback on process improvement in those situations, too. DCA can't be implemented effectively in the total absence of a process framework, but it can help steer the evolution of an organization's process well before it achieves Level 5.

DCA is a low-cost strategy for improving software quality that has proven effective in many different organizational settings. Although the benefits of DCA take time to realize, implementing the project-level approach I've described typically produces measurable results within months. Of course, without timely follow-through by the action team, you will obtain no benefits at all.

This approach to DCA is based on sampling, not an exhaustive analysis of all reported problems. A systematic error is likely to be represented in a given sample, or the next. Actions should focus on the systematic errors—a subset of the sample. Some organizations have conducted causal analyses of 200 problems and produced 500 proposed actions. That quantity of recommendations is likely to overwhelm management's ability to react. Effective action is more likely if you propose a few, high-leverage actions.

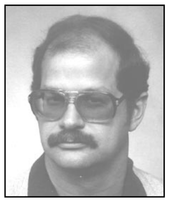
DCA is readily adapted to tasks besides software production. For example, the goal of testers is to find all defects before the system is fielded. Any problem reported from the field represents a defect for the testers. This information can be used in the DCA process to identify testing-process improvements that could facilitate finding those defects before they are fielded.

Many of the actions proposed by the DCA teams will be small incremental improvements rather than revolutionary ideas. Unfortunately, we devote much of our professional lives to dealing with problematic trivia—small things that go wrong, but take up lots of time. Reducing these hindrances lets an organization perform at its true potential and perform actions that may have dramatic effects. ❖

References

1. R. Mays et al., "Experiences with Defect Prevention," *IBM Systems J.*, Vol. 29, No. 1, 1990, pp. 4-32.
2. O. Dangerfield et al., "Defect Causal-analysis: A Report from the Field," *Proc. ASQC 2nd Int'l Conf. Software Quality*, ASQC, Milwaukee, Wisc., Oct. 1992.
3. K. Ishikawa, *Guide to Quality Control,* Asian Productivity Organization, Tokyo, 1976 (revised 1982).
4. R. Chillarge et al., "Orthogonal Defect Classification," *IEEE Trans. Software Eng.*, Nov. 1992, pp. 943-955.
5. M. Paulk et al., "Capability Maturity Model, Version 1.1," *IEEE Software*, July 1993, pp. 18-27.
6. D. Card, "Defect Causal Analysis Drives Down Error Rates," *IEEE Software*, July 1993, pp. 98-99.

About the Author



David N. Card is the manager of the software measurement program at the Software Productivity Consortium. He previously worked for Lockheed Martin, Software Productivity Solutions, and Computer Sciences Corporation. He was a resident affiliate at the Software Engineering Institute and spent seven years as a member of the NASA Software Engineering Laboratory research team.

Card received a BS in interdisciplinary science from American University. He is a member of the IEEE Computer Society and the American Society for Quality.

Address questions about this article to Card at 115 Windward Way, Indian Harbour Beach, FL 32937; card@software.org.