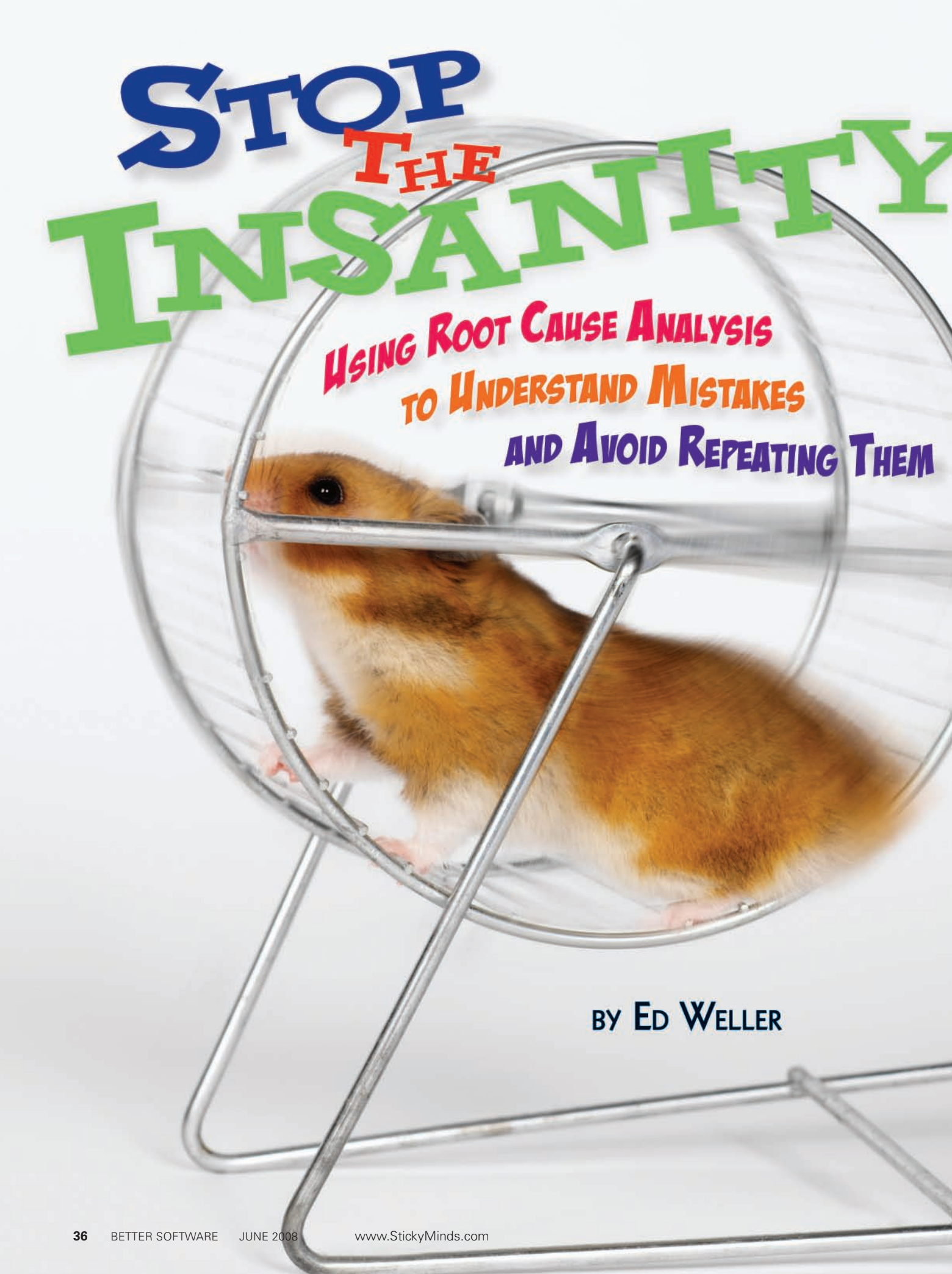
# Stop the Insanity

## Using Root Cause Analysis to Understand Mistakes and Avoid Repeating Them



BY ED WELLER

"To err is human, to forgive is divine."
— Alexander Pope

"Insanity: doing the same thing over and over again and expecting different results."
— Albert Einstein

These two statements summarize the underlying philosophy of root cause analysis (RCA). We must accept the fact that we make mistakes in product development, but we must also accept the responsibility for not repeating those mistakes. Absent any proactive steps to prevent mistakes, we won't be very successful in preventing them. And absent any proactive steps to understand our mistakes, we will repeat them, thus meeting Einstein's definition of insanity.

RCA has many definitions. To some, it is identifying the cause of an application failure and fixing it. To others, it is identifying the reasons for making the initial error or mistake that led to the failure. This confusion is in part caused by differences in analyzing and preventing physical failures versus analyzing and preventing errors in the intellectual activity used in developing software. Industrial accidents or system failures often have an unknown physical cause until investigated, and the RCA focuses on identifying the mechanical, chemical, electrical, or physical cause of the failure. These include mechanical stress, overheating, fatigue, etc. Prevention usually requires a change to one or more physical components of the system. On occasion, these failures are traced back to design or management decisions.

When we trace software failures back to the initial mistake, most often they are the result of an error in our thought processes or intellectual activities. In software development, the common view of RCA is identification of conditions or events that caused a person or team to make an initial error that later manifested as a defect or failure.

To further understand this difference, let's consider some well-known failures. There have been two space shuttle disasters. In both cases, the manifestation of physical failure had been observed multiple times in earlier flights. In both cases, management decisions did not correctly assess the risk of catastrophic failure. These disasters occurred through a combination of physical faults and decision-making faults. The TWA800 disaster was caused by multiple faults: an explosive air-fuel mixture in a fuel tank and an ignition source. The older model F-15 fighters are failing due to stress fractures. These two cases can be attributed to identifiable chemical or mechanical causes that were not identified or understood in the design of these systems. In all four cases, RCA could stop at the "how to fix it" point or, as in the shuttle case, go beyond the physical cause to the underlying decision process.

However, when we consider software, we have a situation where trivial oversights or typos can cause expensive failures. A missing hyphen caused a $500 million Mariner probe to Venus to malfunction. A one-character error in the AT&T 5ESS system caused a $1 billion failure in the 800 phone system. On the other hand, one-character errors in software often are discovered in reviews, testing, or even in use with insignificant failure costs. The lack of a relationship between the error significance and the failure cost makes software RCA fundamentally different. The same trivial error can lead to failures with orders of magnitude of difference in cost. Or, looking at this from another viewpoint, identifying the root cause of a costly failure does not guarantee you will eliminate costly failures elsewhere due to similar causes. In all of these cases, the programmer made an error in creating the work; in other words, the intellectual thought process failed. In software work, there is no physical trail of evidence to lead us to the error. In addition, we have the problem of the frequency of errors. Any sizeable system will have hundreds of errors waiting for the right set of conditions to cause a failure.

Let's establish some definitions. The first three are adapted from *Software Metrics* by Fenton and Pfleeger:

- Human error—a mistake by a human in developing a software work product
- Fault—the encoding of the human error into the software; note that one error may result in multiple faults
- Failure—the manifestation of the fault in the execution of the software
- Defect—often used to mean any or all of the above, but in my opinion it's best used to refer to faults or failures discovered during reviews or testing

Software failures and defects are discovered in use or by testing. At times, the failures require extensive analysis to pinpoint the fault in the code. This problem solving is sometimes referred to as RCA. From the perspective of the operational staff or users, this definition works since they do not want the failure to occur again in the existing product; they want it fixed.

From a development perspective, we want to prevent errors due to this root cause from recurring in future products or development activities. This means we must look for the underlying situation that allowed us—or even encouraged us—to make the error. This can be much more difficult than identifying a physical cause. It also means that to fully understand how the error was made, we need information from the person who made the mistake. Others may guess at the cause, but in most cases only the person making the error can provide insight into the intellectual process that initiated the error.

## Common Problems with RCA

I have seen companies try to apply RCA to production failures in an attempt to improve operational reliability over the

**If the results of an RCA are used to reprimand or punish a person who has made an honest mistake, the process will quickly cease to be effective.**

short term. Their thinking seems to be "If we can just get the developers to make fewer errors, we can improve product quality." While this is true over the long run, this approach ignores the time lag between understanding the root cause, identifying where the cause actually occurred, and then applying prevention to that activity. In the case of requirements errors, the next opportunity to apply the prevention is in the requirements gathering activity of the next version or even the next product. This means that actual improvement seen by users will be at least a full product release cycle away.

A second problem is failure to take action after the analysis has identified the root cause. The process seems to morph from "Find the root cause and prevent its recurrence" to "Hold the RCA meeting to meet a goal" and no further time, people, or money are spent in actually preventing the problem from recurring. Eventually this leads to "Why are we wasting time in this meeting? No one ever does anything with the findings," and RCA is abandoned as ineffective.

A third and even deadlier problem is punishing the person who made the error. If the results of an RCA are used to reprimand or punish a person who has made an honest mistake, the process will quickly cease to be effective. Making the same error over and over after identifying the problem and applying corrective action is another story. The key is to use the results of RCAs to improve a person's or team's capability. I would advise erring on the side of "forgiving and learning" rather than "reprimanding and punishing."

## RCA Versus Retrospectives

While an important process improvement technique, retrospectives do not typically focus on root causes. They are a good source of information for RCA but often only identify things that worked well and should be done again or things to be avoided the next time. For example, a finding of a retrospective might be "Too many defects found in module xyz in system test delayed delivery." At this point, an RCA of the cause of those defects would be performed. Another finding might be "Initial estimates were 50 percent too optimistic." An RCA of the estimating process could be conducted. In many cases, the output of retrospectives or lessons learned is identification of the "primary effect" (a term to be explained later) rather than the root cause.

## The RCA Process

The following three methods have been found useful for conducting RCAs for software failures:

- Apollo method
- Fishbone or Ishikawa diagrams
- 5 Why

I prefer the Apollo method for analyzing software failures, as it has *enough*

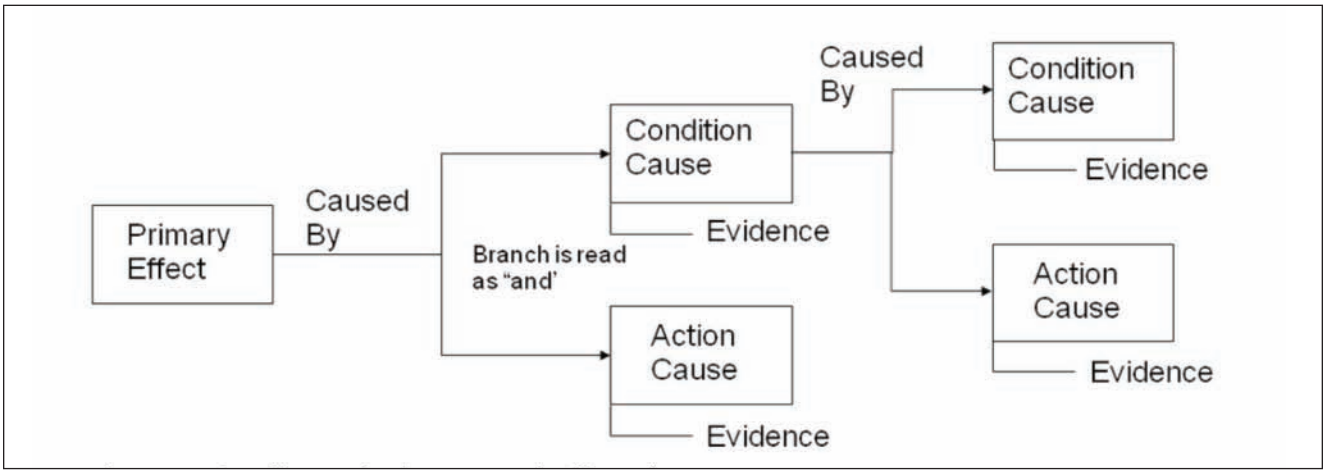



**Figure 1: Apollo method cause-and-effect chart**

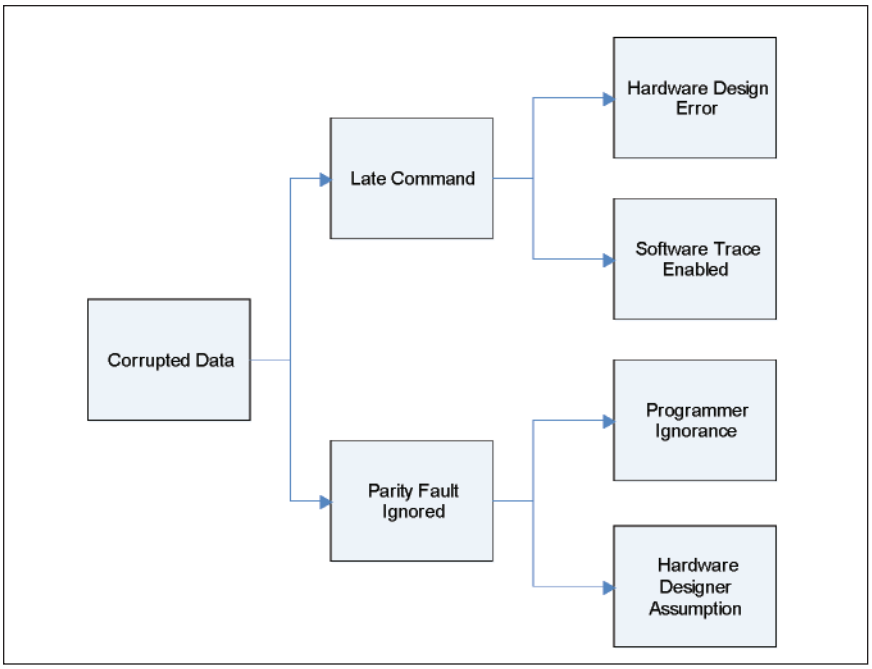



**Figure 2: RCA cause-effect diagram for corrupted data primary effect**

structure but not too much. Fishbone analysis is best left to aggregations of RCAs or process problems. Apollo and 5 Why use similar approaches, but the Apollo method has more structure and guidance. For those wishing to dig deeper into this method, I suggest reading *Apollo Root Cause Analysis* by Dean L. Gano. We'll cover the basic process here.

The Apollo method uses a cause-and-effect chart. The two-level picture shown in figure 1 will be used to explain the principles.

The "primary effect" is the problem or failure we want to prevent from recurring. We need to identify the what, when, and where as well as the significance of the failure. To ensure all are on the same page, these must be clearly stated and understood by all members of the RCA team before the analysis starts. Understanding the significance is needed to allow the team to make appropriate cost-benefit recommendations for those failures where there is a relationship between cause-and-effect cost. Statements must be specific; "System down" has far less meaning than "One hour of production lost at a cost of $1 million."

Once the primary effect is identified, understood, and agreed upon, the next step is to diagram the cause-and-effect chain. In the Apollo method, the term "caused by" is used, and I suggest using this exact phrase when building the chart. Saying "caused by" focuses participants on understanding the link between the two causes or actions. The causes can be divided into a "condition cause" (the necessary set of conditions that exist over time) and an "action cause" (the event that triggered the failure). A software example of a condition cause might be the failure to check for buffer overflow and the action cause a hacker overflowing the buffer in an attempt to breach the system, leading to a primary effect of a security failure. Another condition cause might be programmers failing to monitor device status for data errors and the action cause a reported parity error, leading to the primary effect of corrupted data accepted by the system.

The pairing of the condition cause and action cause is an "and" condition, in which both must exist to cause the primary effect. If either alone could cause the primary effect, then it is likely there are two separate paths, and the causes should be further divided into condition and action causes. Understanding there is a difference may allow the analysis to proceed to a better understanding of the failure cause.

The Apollo method strongly suggests that evidence for the causes be noted. Supposition and guesswork should not be used at this point. If you do not know, close the path with a "?" and go on to examine other paths. Additional investigation should lead to stopping the path or further diagramming when more information is available. The analysis continues down the path until the point of "collective ignorance" is reached. In other words, no one can identify additional caused-by relationships.

Let's analyze a corrupted-data primary effect. In this example, the corrupted data was caused by the action of hardware and software timing (action cause) and software ignoring the parity fault (condition cause). Following the action cause, the timing was caused by a long delay in sending a command (action cause) and a hardware design error (condition cause), caused by enabling the software trace to log events. Following the first condition cause, ignoring the parity fault was caused by programmers unaware of the significance of the fault ("We don't know what to do with parity faults, so we ignore them"). Their ignorance could be traced to lack of knowledge in handling faults and an assumption by the hardware designer that the programmers would "obviously" know a parity error meant data was invalid. Figure 2 shows one way of drawing these relationships. Each branch should be read as "caused by," with the action and condition causes as stated above. I have left the evidence off because in this analysis each of the causes was agreed to by the team and the level of complexity did not require this step.

Note that in this example the first branch is an "and" condition because if either path is prevented, the failure will not repeat.

If we wished, the "hardware design error" path could be further expanded.

As we develop this chart, it is useful to follow the path from left to right, stating "Corrupted data is *caused by* a late command *and* parity fault ignored; late command is *caused by* hardware design error *and* software trace enabled; and parity fault ignored is *caused by* programmer ignorance *and* hardware designer assumption." This verifies that the "caused by" chain is valid and that we have not skipped any intermediate causes.

We should also walk the chart from right to left, stating: "Programmer ignorance and the hardware designer

assumption that programmers knew what to do with parity faults *caused* a parity fault to be ignored, which *caused* corrupted data." This two-way verification of the analysis keeps the logic of the analysis on track.

Once the analysis is finished we are only about 25 percent done! The next step is identifying potential solutions. These should be in line with the resources pre-allocated by management for problem resolution, although there should be exceptions when a serious problem would require more than the agreed upon resources (in other words, although there may be guidelines, let common sense rule the decision).

In the example above, there are four possible solutions:

1. Fix the hardware design error.
2. Do not enable the software trace.
3. Train software programmers on the correct response to the error status.
4. Train the hardware engineer to fully communicate the impact of status errors.

In this case, solutions 1 and 2 *do not* address the root cause. These are problem fixes. If the hardware-design error cause had been traced further, we might get to a root cause (designer oversight, inability to add sufficient hardware to address problem, etc.). To prevent the problem from recurring in other applications with other hardware, solutions 3 and 4 are preventive actions. Adding solutions 3 and 4 to a checklist used at project startup will help the staff "remember" to do this, as well.

## Multiple Defects

Most software systems have a large number of defects. How can root cause analysis be effective in this situation? Typically, causal analysis waits until multiple defects have been found to allow the RCA team to work through ten to twenty defects in a single meeting. This may seem like a large number, but most software defects can be analyzed fairly quickly. A team of developers would wait until a sufficient number of defects have accumulated, perform the RCA for each defect, and then group the causes to identify common causes, if any. If we are able to identify one cause contributing four to six of ten defects, one action can prevent a large percentage of the errors. This is where the real power of RCA comes into play. Carrying this a step further, there should be a team responsible for coordinating results from multiple RCA teams, as an infrequently identified cause at the team level may have a wider impact across the organization.

## Summary

The most important factors in the success or failure of root cause analysis are attitude toward the findings of the causes ("fix the problem" or "punish the people") and follow through with action plans and solutions. If these are done properly, organizations can work through the process. No matter how well the analysis is done, lack of follow through will make the process another make-work task with no value. With the right support and training of participants, root cause analysis can play a significant role in improving product quality and organizational effectiveness. {end}