EuroSTAR eBook
2017 Series

Observing and Reasoning About Errors

Chapt 1:
Why Software Gets In Trouble

Jerry Weinberg

Writer - Consultant (U.S.A)

EuroSTAR Software Testing

biography

contact info:

@JerryWeinberg

Gerald M. Weinberg (Jerry) writes "nerd novels," such as The Aremac Project, about how brilliant people produce quality work. Before taking up his science fiction career, he published books on human behavior, including Weinberg on Writing: The Fieldstone Method, The Psychology of Computer Programming and an Introduction to General Systems Thinking.

Jerry is a host of the AYE Conference (ayeconference.com),winner of both the Warnier Prize and the Stevens Award for his writing on software quality, and also a charter member of the Computing Hall of Fame in San Diego. His website and blogs may be found at http://www.geraldmweinberg.com.

EuroSTAR Software Testing

abstract

People who use software (and builders, too) want to know 'Why Software Gets In Trouble'. You might wonder why anyone would need an entire book on that subject. Why not just say "people make mistakes"? Why not? Because there are reasons people make mistakes, and make them repeatedly, and fail to discover and correct them. Weinberg examines what are software faults and how the organization deals with them means to how the people work. Weinberg helps the reader step back from developing software and examine the dynamics and patterns of software creation. By discussing patterns of quality, patterns of managing and patterns of software faults, the author shows that quality software begins with keen observation and clear thinking about software development.

EuroSTAR
Software Testing

Content

Part IV. Fault Patterns

Three of the great discoveries of our time have to do with programming: the programming of computers, the programming of inheritance through DNA, and the programming of the human mind (psychoanalysis). In each case, the idea of error plays a centralrole.

The first of these discoveries was psychoanalysis, with which Sigmund Freud opened the Twentieth Century and set a tone for the other two. In his introductory lectures, Freud opened the human mind to inspection through the use of errors—what we now call "Freudian slips."

The second of these discoveries was DNA. Once again, key clues to the workings of inheritance were offered by the study of errors, such as mutations, which were mistakes in transcribing the genetic code from one generation to the next.

The third of these discoveries was the stored program computer. From the first, the pioneers considered error a central concern. von Neumann noted that the largest part of natural organisms was devoted to the problem of survival in the face of error, and that the programmer of a computer need be similarly concerned.

In all three of these great discoveries, errors were treated not as lapses in intelligence, or moral failures, or insignificant trivialities—all common attitudes in the past. Instead, errors were treated as *sources of valuable information*, on which great discoveriescouldbebased.

The treatment of error as a source of valuable information is precisely what distin- guishes the feedback (error-controlled) system from its less capable predecessors— and thus distinguishes Steering software cultures from Patterns 1 and 2. Organiza- tions in those patterns have more traditional—and less productive—attitudes about the role of errors in software development, attitudes that they will have to change if they are to transform themselves into Pattern 3 organizations.

So, in the following chapters, we'll explore what happens to Pattern 1 and especially Pattern 2 organizations as they battle those "inevitable" errors in their software. After reading these chapters, perhaps they'll appreciate that they can never move to a Steering pattern until they learn how to use the information in the errors they make.

Chapter 1: Observing and Reasoning About Errors

Men are not moved by things, but by the views which they take of them.- Epictetus

One of my editors complained that the first sections of this chapter spend “an inordinate amount of time on semantics, relative to the thorny issues of software failures and their detection.” What I wanted to say to her, and what I will say to you, is that “semantics” are one of the roots of “the thorny issues of software failures and their detection.” Therefore, I need to start this part of the book by clearing up some of the most subversive ideas and definitions about failure. If you already have a perfect understanding of software failure, then skim quickly, and please forgive me.

1.1. Conceptual Errors About Errors

1.1.1. Errors are not a moral issue

“What do you do with a person who is 900 pounds overweight that approaches the problem without even wondering how a person gets to be 900 pounds overweight?” - Tom DeMarco

This is the question Tom DeMarco asked when he read an early version of the upcoming chapters. He was exasperated about clients who were having trouble managing more than 10,000 error reports per product. So was I.

Over fifty years ago, in our first book on computer programming, Herb Leeds and I emphasized what we then considered the first principle of programming:

The best way to deal with errors is not to make them in the first place.

In those days, like many hotshot programmers, I meant “best” in a moral sense:

(1) Those of us who don’t make errors are better than those of you who do.

I still consider this the first principle of programming, but somehow I no longer apply any *moral* sense to the principle, but only an *economic* sense:

(2) Most errors cost more to handle than they cost to prevent.

This, I believe, is part of what Crosby means when he says “quality is free.” But even if it were a moral question, in sense (1), I don’t think that Pattern 3 cultures—which do a great deal to prevent errors—can claim any moral superiority over Pattern 1 and Pattern 2 cultures—which do not. You cannot say that someone is morally inferior because they don’t do something they *cannot* do, and Pattern 1 and Pattern 2 software cultures, where most programmers reside, are *culturally incapable* of preventing large numbers of errors. Why? Let me put Tom’s question another way:

“What do you do with a person who is rich, admired by thousands, overloaded with exciting work, 900 pounds overweight, and has ‘no problem’ except for occasional work lost because of back problems?”

Tom’s question *presumes* that the thousand-pound person perceives a *weight* problem, but what if they perceive a *back* problem. My Pattern 1 and 2 clients with tens of thousands of errors in their software do not perceive they have a serious problem with errors. They are making money, and they are winning the praise of their customers. On two products out of three, the complaints are generally at a tolerable level. With their rate of profit, who cares if a third of their projects have to be written off as a total loss?

If I attempt to discuss these mountains of errors with Pattern 1 and 2 clients, they reply, “In programming, errors are inevitable, but we’ve got them more or less under control. Don’t worry about *errors*. We want you to help us get things out on *schedule*.” They see no more connection between enormous error rates and two-year schedule slippages than the obese person sees between 900 pounds of body fat and pains in the back. Can I accuse them of having the wrong moral attitude about errors? I might just as well accuse a blind person of having the wrong moral attitude about rainbows.

But it is a moral question for me, their consultant. If my thousand-pound client is happy, it’s not my business to tell him how to lose weight. If he comes to me with back problems, I can show him through a diagram of effects how weight affects his back. Then it’s up to him to decide how much pain is worth how many chocolate cakes.

1.1.2. Quality is not the same thing as absence of errors

Errors in software used to be a moral issue for me, and still are for many writers. Perhaps that’s why these writers have asserted that “quality is the absence of errors.”

It must be a moral issue for them, because otherwise it would be a grave error in reasoning. Here's how their reasoning might have gone wrong. Perhaps they observed that when their work is interrupted by numerous software errors, they can't appreciate any other good software qualities. From this observation, they can conclude that many errors will make software worthless—i.e., zero quality.

But here's the fallacy:

Though copious errors guarantee worthlessness, but zero errors guarantees nothing at all about the value of software.

Let's take one example. Would you offer me $100 for a zero defect program to compute the horoscope of Philip Amberly Warblemaxon, who died in 1927 after a 37-year career as a filing clerk in a hat factory in Akron? I doubt it, because to have value, software must be *more than* perfect. It must be *useful to someone*.

Still, I would never deny the importance of errors. First of all, if I did, Pattern 1 and Pattern 2 organizations would stop reading this book. To them, chasing errors is as natural as chasing sheep is to a German Shepherd Dog. And, as we've seen, when they see the rather different life of a Pattern 3 organization, they simply don't believe it.

Second of all, I do know that when errors run away from us, we have lost quality. Perhaps our customers will tolerate 10,000 errors, but, as Tom DeMarco asked me, will they tolerate 10,000,000,000,000,000,000,000,000,000? In this sense, errors are a matter of quality. Therefore, we must train people to make *fewer* errors, while at the same time managing the errors they do make, to keep them from running away.

1.1.3. The terminology of error

I've sometimes found it hard to talk about the dynamics of error in software because there are many different ways of talking about errors themselves. One of the best ways for a consultant to assess the software engineering maturity of an organization is by the language they use, particularly the language they use to discuss error. To take an obvious example, those who call everything "bugs" are a long way from taking responsibility for controlling their own process. Until they start using precise and accurate language, there's little sense in teaching such people about basic dynamics.

Faults and failures. First of all, it pays to distinguish between failures (the symptoms) and faults (the diseases). Musa gives these definitions:

A failure “is the departure of the external results of program operation from requirements.”

A fault “is the defect in the program that, when executed under particular conditions, causes a failure.”

For example:

An accounting program had a incorrect instruction (fault) in the formatting routine that inserts commas in large numbers such as “$4,500,000”. Any time a user prints a number greater than six digits, a comma may be missing (a failure). Many failures resulted from this one fault.

How many failures result from a single fault? That depends on

- the location of the fault
- how long the fault remains before it is removed
- how many people are using the software.

The comma-insertion fault led to millions of failures because it was in a frequently used piece of code, in software that has thousands of users, and it remained unresolved for more than a year.

When studying error reports in various clients, I often find that they mix failures and faults in the same statistics, because they don’t understand the distinction. If these two different measures are mixed into one, it will be difficult to understand their own experiences. For instance, because a single fault can lead to many failures, it would be impossible to compare failures between two organizations who aren’t careful in making this “semantic” distinction.

Organization A has 100,000 customers who use their software product for an average of 3 hours a day. Organization B has a single customer who uses one software system once a month. Organization A produces 1 fault per thousand lines of code, and receives over 100 complaints a day. Organization B produces 100 faults per thousand lines of code, but receives only one complaint a month.

Organization A claims they are better software developers than Organization B. Organization B claims they are better software developers than Organization A. Perhaps they’re both right.

The System Trouble Incident (STI). Because of the important distinction between faults and failures, I encourage my clients to keep at least two different statistics. The first of these is a data base of “system trouble incidents,” or STIs. In this book, I’ll mean an STI to be an “incident report of one failure as experienced by a customer or simulated customer (such as a tester).”

I know of no industry standard nomenclature for these reports—except that they invariably take the form of TLAs (Three Letter Acronyms). The TLAs I have encountered include:

- STR, for “software trouble report”
- SIR, for “software incident report,” or “system incident report”
- SPR, for “software problem report,” or “software problem record”
- MDR, for “malfunction detection report”
- CPI, for “customer problem incident”
- SEC, for “significant error case,”
- SIR, for “software issue report”
- DBR, for “detailed bug report,” or “detailed bug record”
- SFD, for “system failure description”
- STD, for “software trouble description,” or “software trouble detail”

I generally try to follow my client’s naming conventions, but try hard to find out exactly what is meant. I encourage them to use unique, descriptive names. It tells me a lot about a software organization when they use more than one TLA for the same item. Workers in that organization are confused, just as my readers would be confused if I kept switching among ten TLAs for STIs. The reasons I prefer STI to some of the above are as follows:

1. It makes no prejudgment about the fault that led to the failure. For instance, it might have been a misreading of the manual, or a mistyping that wasn’t noticed. Calling it a bug, an error, a failure, or a problem, tends to mislead.
2. Calling it a “trouble incident” implies that once upon a time, somebody, somewhere, was sufficiently troubled by something that they happened to bother making a report. Since our definition of quality is “value to some person,” someone being troubled implies that it’s *worth* something to look at the STI.

3. The words “software” and “code” also contain a presumption of *guilt*, which may unnecessarily restrict location and correction activities. We might correct an STI with a code fix, but we might also change a manual, upgrade a training program, change our ads or our sales pitch, furnish a help message, change the design, or let it stand unchanged. The word “system” says to me that any part of the overall system may contain the fault, and any part (or parts) may receive the corrective activity.
4. The word “customer” excludes troubled people who don’t happen to be customers, such as programmers, analysts, salespeople, managers, hardware engineers, or testers. We should be so happy to receive reports of troublesome incidents *before* they get to customers that we wouldn’t want to discourage anybody.

Similar principles of semantic precision might guide your own design of TLAs, to remove one more source of error, or one more impediment to their removal. Pattern 3 organizations always use TLAs more precisely than do Pattern 1 and 2 organizations.

System Fault Analysis(SFA). The second statistic is a database of information on faults, which I call SFA, for System Fault Analysis. Few of my clients initially keep such a database separate from their STIs, so I haven’t found such a diversity of TLAs. Ed Ely tells me, however, that he has seen the name RCA, for “Root Cause Analysis.” Since RCA would never do, the name SFA is a helpful alternative because:

1. It clearly speaks about faults, not failures. This is an important distinction. No SFA is created until a fault has been identified. When a SFA is created, it is tied back to *as many STIs as possible*. The time lag between the earliest STI and the SFA that clears it up can be an important dynamic measure.
2. It clearly speaks about the system, so the database can contain fault reports for faults found anywhere in the system.
3. The word “analysis” correctly implies that data is the result of careful thought, and is not to be completed unless and until someone is quite sure of their reasoning.

“Fault” does not imply blame. One deficiency with the semantics of the term”fault” is the possible implication of *blame*, as opposed to information. In an SFA, we must be careful to distinguish two places associated with a fault, either of these implies anything about whose "fault" it was:

a. *origin*: at what stage in our process the fault originated
b. *correction*: what part(s) of the system will be changed to remedy the fault

Pattern 1 and 2 organizations tend to equate these two notions, but the motto, "you broke it, you fix it," often leads to an unproductive "blame game." "Correction" tells us where it was wisest, under the circumstances, to make the changes, regardless of what put the fault there in the first place. For example, we might decide to change the documentation—not because the documentation was bad, but because the design is so poor it needs more documenting and the code is so tangled we don't dare try to fix it there.

If Pattern 3 organizations are not heavily into blaming, why would they want to record "origin" of a fault? To these organizations, "Origin" merely suggests where action might be taken to *prevent* a similar fault in the future, not which employee is to be taken out and crucified. Analyzing origins, however, requires skill and experience to determine the earliest possible prevention moment in our process. For instance, an error in the code might have been prevented if the requirements document were more clearly written. In that case, we should say that the "origin" was in the requirements stage.

1.2. Mis-classification of Error-Handling Processes

By the term "error-handling process," we'll refer to the overall pattern that has to do with errors, a pattern which can be resolved into several activities. Once we understand the distinction among these component activities, we'll be able to describe the dynamics of each in a way that will suggest improvement. Characteristically, however, Pattern 1 and Pattern 2 organizations are not very adept at knowing just precisely what their error-handling process is. If you ask, the typical answer will be "debugging." With that sort of imprecise speech, improvement in fault-handling is unlikely.

1.2.1. Detection

Detection of faults is achieved in different ways in different software cultures. Pattern 1 and 2 organizations tend to depend on faults being detected by *failures* in some sort of machine execution of code, such as machine software testing, beta testing, and operational use by customers. These are the STIs.

Pattern 3 organizations also detect faults though failures, but tend to prefer going *directly to faults* by some process that does not require machine execution of the code. These mechanisms include accident (such as running into an error while looking in code for something else), technical reviews of great variety, and tools that process code, designs, and requirements as analyzable documents, suggesting failures without machine execution of the code itself. These methods result in SFA which don't necessarily correspond to any STI, if they were applied early enough to prevent any failure resulting from the fault.

1.2.2. Location

Location, or *isolation*, is the process of matching failures with faults. Even when a fault is found directly, as in a code review, good practice dictates that the SFA contain a trace forward into the set of failures known to exist. Only by forward tracing can unsolved failures be cleared out of the STI data base. If a great many unsolved STIs remain, managers and programmers have a tendency to discount all of them, which makes location of truly active STIs more difficult.

1.2.3. Resolution

Resolution is the process that ensures that a fault no longer exists, or that a failure will never occur again. A failure may be solved without having its fault or faults removed. Removal of faults is an optional process, but resolution is not. Resolution of an STI may be performed in several ways:

1. Remove the fault that led to the STI. This is the "classic" way of "debugging."
2. Define the STI as unimportant, such as "too minor to fix," or "non-reproducible."
3. Define the STI as not arising from a fault in the system, but usually as a fault in the person who reported it.

4. Define the fault as not a fault, such as by following the Bolden Rule that says, "If you can't fix it, feature it."

In troubled Pattern 2 organizations, the majority of STIs are resolved by 2, 3, and 4, while management believes they are resolved by (1).

1.2.4. Prevention

Prevention may seem a pie-in-the sky approach to people buried deep in Pattern 1 and Pattern 2 organizations. The history of other engineering disciplines assures us that some schemes for preventing errors will ultimately prevail, but these seem a long way from where most of my clients are standing today. When I show them articles about Pattern 3 organizations, they say they're not applicable to their organizations. When I show them articles about "clean room" software development or other Pattern 4 techniques, they simply chuckle with disbelief.

In fact, however, most of the error work in a software development organization is actually prevention work, though Pattern 2 managers don't understand this. Only after they become rather sophisticated in analyzing software engineering dynamics do they realize that most of their activities are in place to prevent error, not fix it. Just to take one example, ask people why they follow the practice of "design before code." Very few of them will recognize this rule as an error-prevention strategy dictated by the war against the Size/Complexity Dynamic.

1.2.5. Distribution

In Pattern 2 organizations, *distribution* of errors is an important and often time-consuming activity. By distribution, we mean any activity that serves to prevent attributing errors to one part of the organization by *using the technique of moving them to another place*. Here are some examples of distribution:

- Developers quickly throw code "over the wall" to testers, so that errors are seen as somehow arising during test, rather than fromcoding.
- The organization skips the design reviews so that design faults are seen as coding faults.

- Testers pass code into operations so problems can be classified as "maintenance faults."

These three examples are the type of distribution activity that prevents *blame*, in response to measurement systems that are used punitively rather than for control activities. When you don't know how to prevent errors, what else can you do but prevent blame for errors? Of course, to the extent that the workers are playing "hot potato" with faults, they have that much less time to do actual productive work. We'll see more about the hot potato phenomenon when we study the dynamics of management pressure.

But not all distribution activities are disguised forms of hot potato. Where blame is not the name of the game, distribution actually serves useful purposes. Pattern 3 organizations tend to distribute the faults *earlier* in the process than Pattern 2 organizations:

- Design and requirements work are seen as ways of catching large scope faults early in the development process, rather than later when they will be more costly to resolve.
- User manuals are written early as a way of revealing faults in interface requirements and generating the basis for acceptance tests. Once again, this unburdens the late parts of the development cycle.

1.3. Observational Errors About Errors

Failure detection is a process of *noticing differences*—between what is desired and what exists. When we consider the cybernetic model of control, we understand how important seeing differences—failure detection—is to a feedback controller.

Giving things labels is a substitute for noticing. That's another reason I always emphasize the importance of the words controllers use. It's all too easy not to notice important differences if you name two things the same, or to see a difference where none exists if you name them differently.

1.3.1 Selection Fallacies

There is a whole class of common mislabelings which I call “selection fallacies.” A selection fallacy occurs when a controller makes an incorrect linear assumption about observation that says, incorrectly,

“I don’t have to observe the full set of data, because a more easily observed set of data adequately represents it.”

It’s a fallacy because it doesn’t take into account that the processes of selecting the two groups of data may be different, and thus conclusions drawn from one group may not apply to the other. Selection fallacies are easy to spot *after the fact*, but easy to fall into before, especially if we have some reason to *want* one conclusion more than another.

Completed vs. terminated projects. Here’s an example of a common selection fallacy in software:

A client surveyed the number of faults produced per thousand lines of code (KLOC) in 152 projects. The study was done very carefully, using the SFA data base for each project. The study concluded that the average project produced 6-23 faults/KLOC, with an average of 14. They felt that this was in line with other organizations in their industry, so they had no strong motivation to invest in further reductions.

Listening to the presentation of this careful study, it would have been easy to miss the selection fallacy, but always being cautious, I asked, “How did you choose the 152 projects?”

“Oh, we were very careful not to bias the study,” the presenter said. “We chose *every* project that was completed in a 3-month period.”

“You emphasized the wrong word,” I said, now seeing the selection fallacy.

“What do you mean?” he asked.

“You should have said, ‘We chose every project that was *completed* in a 3-month period.’ How many projects here are started that *never* complete?”

The presenter didn’t know, and neither did anyone else in the room. I got them to give an approximation, which was later verified by a small study. Historically, in this organization, 27% of initiated projects were never completed. These projects accounted for over 40% of their development budget, because some were not abandoned for a long, long time. A sample of these projects showed a range of 19-145 faults/KLOC, with an average of 38. Later, when the average was weighted by project *size*, it grew to 86. The two biggest projects also had the highest faults/KLOC.

Where had they gone wrong? In presenting *completed* projects as representative of *all* projects, the presenters had committed a common selection fallacy which led the organization to believe that they were not too bad in their fault-producing performance. Then, when they presented *all failed projects* as typical of their worst projects, they committed the same fallacy in reverse. The second fallacy led them to miss the fact that they simply didn't know how to develop large projects, probably because they couldn't deal with the faults they generated.

Early vs. late users. Here's another common selection fallacy:

A software organization shipped an update to product X and tracked the STIs that arrived in the first two months. They used these to make a linear projection of the STI load they would have to handle in the following months. Their estimate of the number of STIs was quite accurate, but the total workload generated by those STIs was underestimated by a factor of 3.5.

They had committed several selection fallacies, all based on the assumption that early STIs would be typical of later STIs. They were not because,

1. Later STIs had a far higher failure/fault ratio, because more customers were using the system and encountering the same failures multiple times. The company had no efficient way of resolving these multiple reports of the same failure.
2. Early users of the update were not typical of late users. They tended to be more self-reliant, and worked around a number of failures that later users had to report as STIs in order to get help. Although they were easy to work around, their underlying faults were not necessarily easy to resolve.
3. Early users also tended to use a different set of features than the later users. The typical later user was later because their use was much more extensive, both in features covered and number of people having access to the system. These attributes meant that their installation procedure was more complex, thus slower, which is why they were later users. But more people accessing the system, using more of the features, meant many more STIs.

"He's just like me." The selection fallacy works not only on the observations, but also on the *observers*. Here's the continuation of the story about Simon, the project manager who couldn't recognize tears.

After Simon asked me whether there was something in Herb's eye, I said, "Well, I really don't know. Why don't you ask him?"

"Oh, it's not really important enough to take the time," Simon replied. "I need to ask you how you think the project is going? I'm really pleased at what a great job Herb did, getting that program ready just a week late, after it was in so much trouble."

"Really?" I said. "I thought you were rather upset about the late delivery."

"Oh, that. Sure, I'd like to have had it on time, but it's no big deal."

"I think Herb thought it was a big deal."

"What makes you think that?"

"I believe he was upset when you yelled at him."

"Oh, no. Herb knows me too well to be upset, just because I raised my voice a little. He's just like me, so he knows I'm just an enthusiastic guy."

Simon committed a selection fallacy by assuming "he's just like me." The managers in a software organization are not "just like" the rest of the people—else why were they selected to be managers, and why are they being paid more money? Why not observe the other person instead of assuming the two of you are exactly alike?

Any manager who can't or won't see or hear other peoples' feelings is like a ship's captain trying to navigate at night without radar or sonar. Feelings are the radar and sonar of project life—reflections off the reefs and shoals and shallow bottoms on which your project can run aground. You can't do it with your eyes and ears closed, just using a map inside your own head, if only because you're not just like everyone else.

1.3.2. Getting observations backwards

It's one thing to fail to observe something correctly. It's quite another to observe correctly, but then to interpret the observations *backwards*, so that black is labeled white and white is labeled black. Some people have a hard time believing that a highly paid software engineering manager could actually label observations backwards, so here a few examples of hundreds I've observed.

Who are the best and the worst programmers?

A software development manager told me that he had a way to measure who were his best programmers and who were his worst. I was fascinated, so I asked him how he did it. He told me that he sat in his office and observed who was always out asking questions of users and other programmers. I thought this was a terrific measure, and I discussed it with him with great excitement. After a few minutes, however, I realized that he thought the programmers who spend the most time asking questions are his worst ones. I, on the other hand, thought that some of these were probably the best programmers in his shop.

Which is the good quality release?

When she received her first monthly STI summary for a newly released product, the vice-president of software technology waved it at me and said, "Well, we've finally put out a high-quality release." As it turned out, the release offered so little new function that essentially nobody bothered to install it. Hence, there were virtually no troubles reported. Later, when people did have to install it, they found that it was just as full of errors as all the previous releases.

Why is someone working late?

A programming team manager told me, "Josh is my best programmer. The reason he starts work in the afternoon and stays late at night is so he won't be disturbed by the less experienced programmers." It turned out that Josh was so ashamed of the poor quality of his work that he didn't want anyone to see how much trouble he was having.

Who knows what's right and wrong?

Another team leader told me, "Cynthia is angry because I showed her what was wrong with her program, and how it should have been done in the first place. I suppose you're going to tell me I have to learn to be more tactful." Cynthia showed me the program and what the team leader had said was wrong. It wasn't wrong at all. Cynthia said, "What ticks me off is working under a boss who's not only technically illiterate, but doesn't know how to listen. He approaches every problem with an open mouth."

Which process is eliminating problems?

A project manager told me, “We’ve abandoned technical reviews in this project. They were valuable at first, and we found a lot of problems. Now, however, they don’t find much trouble—not enough to justify the expense.” As it turned out, the reason there were no problems was that the programmers were conducting secret reviews, to hide their errors from the manager, who berated anyone whose product showed errors in the review. They had not abandoned technical reviews; they had abandoned the practice of telling their manager about their technical reviews.

Feedback controllers use observations of behavior to decide upon actions to eliminate undesired behaviors. They feed these actions back into the system and thus create a negative feedback loop to stabilize the system, as shown in Figure 1-1.

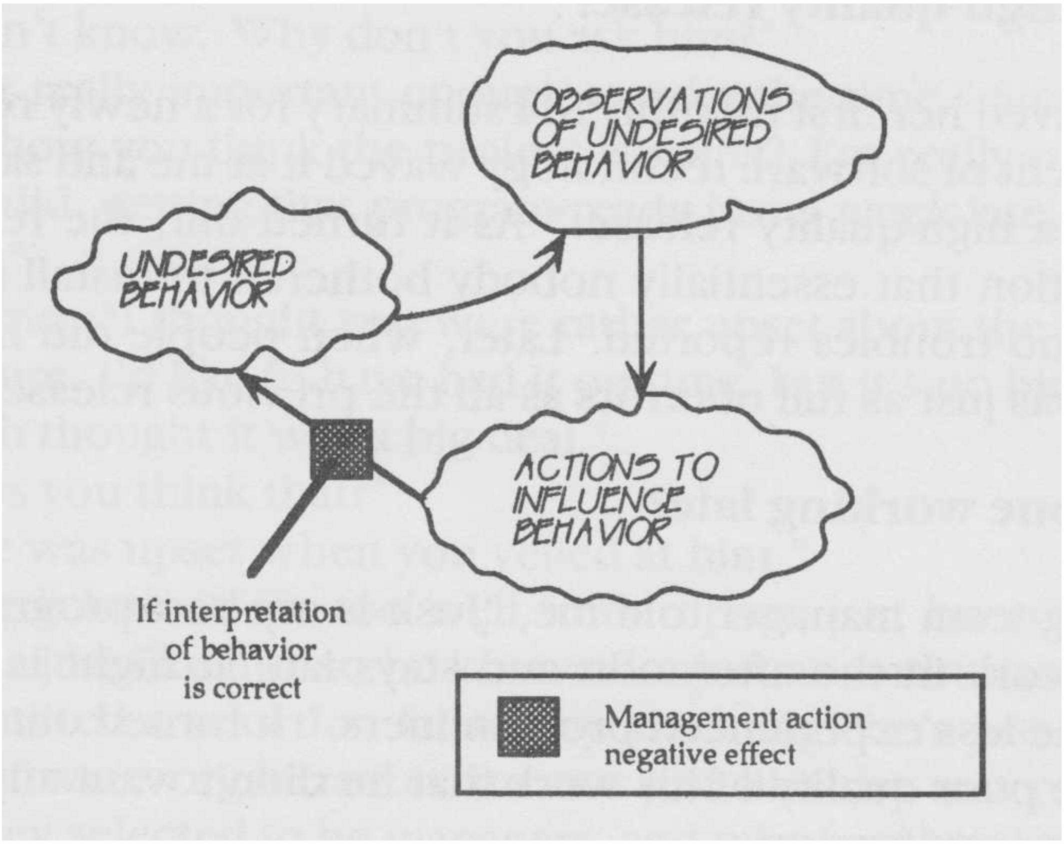
OBSERVATIONS OF UNDESIRED BEHAVIOR
UNDESIRED BEHAVIOR
ACTIONS TO INFLUENCE BEHAVIOR
If interpretation of behavior is correct
Management action negative effect

Figure 1-1. The feedback controller uses observations to decide upon actions to

stabilize the system's behavior.

When the feedback controllers gets the meaning of the observation backwards, however, the designed actions create a positive feedback loop, actually encouraging the undesired behavior, as shown in Figure 1-2.

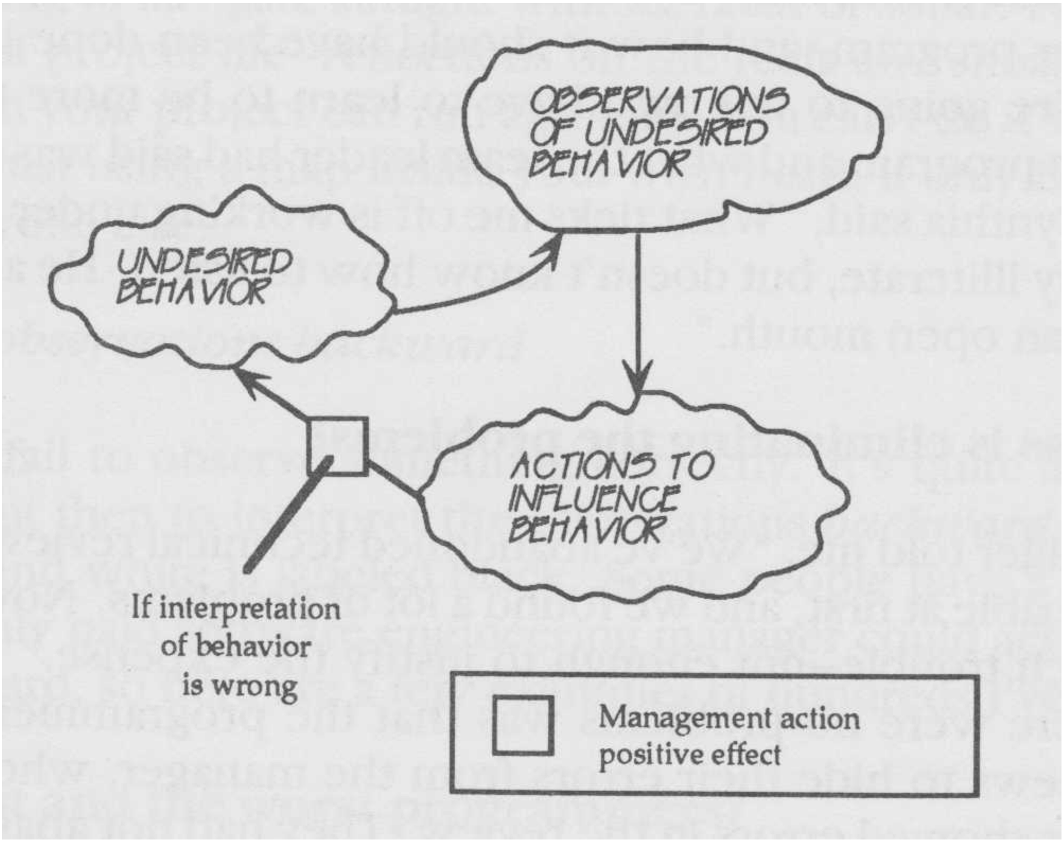
OBSERVATIONS OF UNDESIRED BEHAVIOR
UNDESIRED BEHAVIOR
ACTIONS TO INFLUENCE BEHAVIOR
If interpretation of behavior is wrong
Management action positive effect

Figure 1-2. Getting the meaning of an observation backward creates an intervention loop that promotes what it should discourage and vice versa.

1.3.3. The Controller Fallacy

The example of eliminating technical reviews illustrates another common observational fallacy. Even if it were true that reviews were no longer finding errors, would that be the reason to abandon them? Technical reviews serve many functions in a software project, but one of their principal functions is to provide feedback information to be used in controlling the project. In other words, they are part of the controller's system.

It is the nature of feedback controllers to have an inverse relationship to the systems they attempt to regulate.

- We spend money on a thermostat so we won't spend extra money on fuel for heating and cooling.
- We keep the fire department active so that fires will be inactive.
- We put constraints of the powers of government so that government won't put unnecessary constraints on the governed.

One result of this inverse relationship is this

The controller of a well-regulated system may not seem to be working hard.

But managers who don't understand this relationship often see lack of obvious controller activity as a sign that something is wrong with the control process. This is the Controller Fallacy, which comes in two forms:

If the controller isn't busy, it's not doing a good job. (NOT)

If the controller is very busy, it must be a good controller. (NOT)

Managers who believe the second form are the ones who "prove" how important they are by being too busy to see their workers.

The first form applies to the reversed technical review observation. If the technical reviews are not detecting a lot of mistakes, it could mean that the review system is broken. On the other hand, it could also mean that the review system is working very well, and preventing faults by such actions as

- motivating people to work with more precision
- raising awareness of the importance of quality work
- teaching people how to find faults before coming to reviews
- detecting indicators of poor work, before that work actually produces faults
- teaching people to prevent faults by using good techniques they see in reviews

1.4. Helpful Hints and Suggestions

- Usually, there are more failures than faults, but sometimes, there are faults that produce no failures—at least given the usage of the software up until the present time. And sometimes it takes more than one fault to equal one failure. For instance, there may be two that are "half-faults," neither of which would cause trouble except when used in conjunction with the other. In other cases, such as in performance errors, it may take an accumulation of small faults to equal a single failure. This makes it important to distinguish between functional failures and performance failures.
- Proliferation of acronyms is a sign of an organization's movement towards Pattern 2, where name magic is so important that a new name confers power on its creator. Care in designing acronyms is a sign of an organization's movement towards Pattern 3, where communication is so important.
- You can almost count on the fact that first customers aren't like later ones. Managers often commit a selection fallacy in planning their future as software vendors based on initial favorable customer reactions to a software system. The first customers are first because they are the ones the requirements fit for. Thus, they are very likely to be "like" the original designer/developers. Developers and customers communicate well, and think alike. This is not so as the number of customers grows, and explicit processes must be developed to replace this lost "natural" rapport.
- Selection fallacies are everywhere. You can protect yourself by wearing garlic flowers around your neck, or else by asking, whenever someone presents you with statistics to prove something, "Which cases are in your sample? Which cases are left out? What was the process by which you chose the cases you chose?

1.5 Summary

1. One of the reasons organizations have trouble dealing with software errors is the many conceptual errors they make concerning errors.
2. Some people make errors into a moral issue, losing track of the business justification for the way in which they are handled.

3. Quality is not the same thing as absence of errors, but the presence of many errors can destroy any other measures of quality in a product.
4. Organizations that don't handle error very well also don't talk very clearly about error. For instance, they frequently fail to distinguish faults from failures, or use faults to blame people in the organization.
5. Well functioning organizations can be recognized by the organized way they use faults and failures as information to control their process. The System Trouble Incident (STI) and the System Fault Analysis (SFA) are the fundamental sources of information about failures and faults.
6. Error-handling processes come in at least five varieties: detection, location, resolution, prevention, and distribution.
7. In addition to conceptual errors, there are a number of common observational errors people make about errors, including Selection Fallacies, getting observations backwards, and the Controller Fallacy

1.6. Practice

1. Here are some words I've heard used as synonyms for "fault" in software: lapse, slip, aberration, variation, minor variation, mistake, oversight, miscalculation, blooper, blunder, boner, miscue, fumble, botch, misconception, bug, error, failure. Add any words you've heard to the list, then put the list in order according to how much responsibility they imply on the human beings who created the fault.
2. When an organization begins the systematic practice of matching every failure with a known fault, it discovers that some failures have no corresponding fault. In Pattern 3 organizations, these failures are attributed to "process faults"—something wrong with their software process that either generated fictitious failures or prevents the isolation of real ones. List some examples of process faults commonly experienced in your own organization, such as careless filling out of STI records.
3. For a week, gather data about your organization in the following way: as you meet people in the normal course of events, ask them what they're doing. If it has anything to do with errors of any kind, make a note of how they label their activity—debugging, failure location, talking to a customer, or whatever.

At the end of the week, summarize your findings in a report on the process categories used for error work in your organization's culture.

4. Describe a selection fallacy that you've experienced. Describe its consequences. How could a more appropriate selection have been made?

promotion

EuroSTAR Software Testing CONFERENCE

copenhagen 06-09 NOVEMBER 2017

10 % DISCOUNT

In 2017 we celebrate our 25th annual conference – from 6-9 November – at the Bella Center in Copenhagen (Denmark).

Until **September 22nd** you chance to avail of a 10% discount on all tickets. Click the button below for details

get discount code

EuroSTAR Software Testing

want more?

Enjoyed this eBook and want to read more?

Check out our extensive eBook library on Huddle.

EuroSTAR Software Testing | Huddle

Join us online at the links below

EuroSTAR Software Testing

purchase

The full version of this eBook and book is now available to purchase from:

www.amazon.com

www.Leanpub.com

buy now

EuroSTAR Software Testing

www.eurostarsoftwaretesting.com

EuroSTAR Software Testing