Destroying Communication and Control in Software Development©

Dr. Gerald M. Weinberg
Weinberg & Weinberg

More than half of large systems-development projects fail, either intentionally or not, because people destroy communication and control within the organization. This article explains how this occurs when using basic tools, including requirements, configuration management, testing, and more. Fortunately, as the author reveals the methods used to sabotage these tools, he also advises project managers on where and when to intervene with countermeasures.

More than half of large systems-development projects fail, and of those that succeed, very few are delivered on schedule [1]. The situation is so bad that nontechnical executives often ask, "How does a project get to be one year late?" Years ago, Fred Brooks gave the astute answer, "One day at a time" [2].

Astute as it was, Brook's answer was not too helpful for those trying to teach nontechnical executives how to avoid these hyper-extended projects – or at least to see them coming. What exactly *happens* on those days where the project falls behind? Watching the progress of the war in Afghanistan, I finally realized how to explain this dynamic so everyone could understand.

In modern warfare, the first step in hostilities is to destroy the enemy's communication and control system, after which they can be easily defeated because of their confusion and inability to coordinate their forces. This is exactly the strategy that seems to be taking place when the development organization destroys its management's ability to control the organization. It may or may not be intentional. It may not happen all at once. But one day at a time, one small step at a time, the net effect is the same as a carefully planned and executed war – destroying the manager's ability to manage successful projects.

The *war* in this case is a war against *nature*, including human nature. A software product is, in essence, composed of millions, or tens or hundreds of millions of tiny *parts*. Each of those parts must be built correctly, but that is not enough. Building the product consists of putting those correctly built parts together in the proper sequence. If we consider each part as analogous to a military target, the desired product is a large set of targets. When all the targets have been hit, in the proper sequence, the war is won and the product is built successfully.

But by nature, human beings are not well equipped for such precision work. Errors (missed targets) occur in every product development. Unless a project is well managed, the war is never won, and the product is never finished or is finished poorly.

Why would a project be managed badly? It is an application of the *first law of bad management*: "If what you're doing isn't working, do more of it" [3]. I used to wonder how managers could be so stupid as to continue doing things that were not working. After watching many software development wars, I realized that it is not a matter of stupidity. As their communication and control system is destroyed, managers simply *do not know* that what they are doing is not working. They do not know that essential targets are being missed.

> *"As their communication and control system is destroyed, managers simply do not know that what they are doing is not working."*

Suppose you were a military general directing air strikes by using video cameras to show the target area. Suppose those cameras were not actually showing the target area, but recorded scenes from some other area. You would continue bombing, thinking you were hitting targets until the enemy mustered a surprise attack with forces you thought had been eliminated.

Managers whose communication and control systems have been corrupted usually do not know anything is wrong until the delivery date arrives. The delivered code is not complete, and what is complete is full of errors that have not been eliminated. If the communication and control system has not been destroyed but instead *rigged* to fool them, they may not even know that they do not really have the product they expected to have. In the following sections, I will cover five general categories of communication and control disruption:

- Destroying information.
- Destroying information infrastructure.
- Hiding information.
- Degrading the believability of information.
- Inserting misleading information.

Let me emphasize that these disruptions do not have to be intentional, although they might be. People with the very best intentions, even the managers themselves, may do each of them. That is because people are not perfect data recorders. They are influenced by stress, by pressure, by what they think will happen to themselves or other people, and by just plain mistakes. Only in the smallest of projects can a manager rely solely on personal reports by individuals, no matter how well intentioned they may be.

Instead, as projects grow larger, managers must create and guard communication and control systems that protect against disruption by individual error; these systems must in turn be protected from disruption. I will illustrate such disruptions by examining typical mistakes when using the following principal tools that managers require for communication and control:

- Requirements will be used to illustrate destroying information.
- Configuration management will be used to illustrate destroying of information infrastructure.
- Technical reviews, project management reviews, and quality assurance will be used to illustrate hiding information.
- Testing will be used to illustrate degrading the believability of information.
- Demonstrations and risk management

© 2002 Gerald M. Weinberg. All Rights Reserved.

processes will be used to illustrate inserting misleading information.

Of course, the mistakes are not limited to the following examples, and each type of mistake can be found in each communication and control tool.

Destroying Information

In software systems, the product is invisible unless special efforts are made to render it visible. To take a simple example, an individual programmer might make 50 test-runs but never record that fact, or tell anybody. He might find a dozen faults with his code that he does not fix and never mentions. Pair programming (as in eXtreme programming) is one way to prevent this kind of information destruction, but not if both members of the pair collaborate in the destruction.

In project reviews, we frequently see another common form of information destruction: the shading of figures. To take a common example, a piece of work on the critical path is a week behind; however, the project manager makes a *slight adjustment* so that this delay will not raise an issue with upper management. After all, he or she reasons, everyone will just work a little harder and catch up. Maybe they will, and maybe they will not, but now the manager is steering by a distorted *video* rather than by a view of the actual battlefield.

Requirements

Because software is an invisible product, The Zero Law of software engineering states, "If you don't have to meet quality requirements, you can meet any other objective" [4]. Thus, whenever there are not clear requirements on quality, anybody can say, "We're doing just fine." It is like dropping bombs without a map of target locations – all you can say is that you have dropped a lot of bombs, not what effect it is having on the enemy. So, without defined requirements, so-called progress reports are not reports of progress at all but merely reports of effort expended – the number of bombs dropped in the dark, and the number of days working with unknown results.

Without requirements, when the software finally becomes executable and something bad happens the developer can say, "That is not a bug, it is a feature." ("That hospital we destroyed was a secret enemy base.") By preventing requirements from being explicit, developers can thus ruin any real information about quality.

But how is this done when everybody knows the importance of requirements? Early in a project when a manager tries to obtain explicit requirements, he or she will hear dozens of excuses, some of the most common of which are these:

- "We know what is needed, so writing it down will be a waste of time."
- "It is too hard to get everyone to agree. It just creates conflict."
- "You cannot really know the requirements until you let the customer see it, so we should not bother them."
- "It will take a long time, and we have to start the real work of coding, otherwise we will not meet the schedule."

Each such argument has some appeal for the manager who does not want to waste time, create conflict, bother customers, or miss the schedule. However, in accepting such arguments, the manager self-destroys his or her ability to communicate and control. In doing so, he or she ensures that time will be wasted, conflict will be rife, customers will be bothered, and the project will not meet its schedule – exactly the opposite of the intended effects.

> *"By preventing requirements from being explicit, developers can thus ruin any real information about quality."*

So why do managers keep falling into this self-destructive trap? I believe it is because there is some truth to each argument: Requirements work, when done badly, can waste time, engender conflict, irritate customers, and delay a project. The solution, though, is not to eliminate requirements work and rush into coding, but to create and support an effective requirements process. There are a number of ways to do this, and the manager must not fall into the trap of indecision about which one to use.

Choose a process for managing both initial requirements and changes to requirements. Train people or hire experienced people to execute your process. Above all, see that the process is actually carried out. Only then can you have fact-based communication and control. Otherwise, you will be like an artillery commander whose gunners report they are never missing the intended target, which is technically true because nobody specified a target.

Destroying Information Infrastructure

High-level managers can generally evaluate the success of a requirements system because they should be able to understand the targets – those features and attributes that their customers desire, at least the nontechnical ones. But high-level managers are generally not qualified to examine technical details and determine their correctness. Even if qualified, certainly they have no time for the job. Instead, they must rely on indirect information about quality beneath the requirements level.

The principal management tools for obtaining such information in an understandable form are personal reports from the technical staff plus reports from quality assurance, technical reviews, and testing. These are the radar detectors, reconnaissance satellites, cryptographers, and field reports in the war against error. Anything that deranges or distorts personal reports, quality assurance, technical reviews, or testing will destroy essential communication and control information.

Underlying all these reporting systems is the configuration management system (CMS) that has the job of retaining all essential project information: requirements, design, code, test plans, test data, test results, review reports, project management information, architectural data, and user documentation.

The CMS is designed to prevent physical destruction of information such as altering of reports; unrecorded changes to requirements, code, or tracking data; unauthorized entries in data fields; or physical failure of media. Without a functioning CMS, the manager cannot rely on the accuracy of any information. However, the CMS is easily undermined in numerous ways that illustrate the destruction of the information infrastructure.

The CMS

Perhaps the most common way to destroy a CMS is by passive behavior. People do not object directly to the system, but they fail to use it or fail to use it correctly in ways that might be attributed to innocent misunderstandings.

For instance, items that are supposed to be placed in the CMS are not found there. When someone asks the responsible people, they might say, "Oh, we did not know you needed that in draft form. We still have some unresolved issues, so we thought we would wait until it was perfect before we put it in." It is difficult for a manager to fault people who merely say

they are trying to do a good job.

To take another example, consider the bug-tracking database, which is part of the CMS that is supposed to report each error found in testing. Each error report remains *open* until the error is tracked down and repaired. Successful managers rely on statistics from the bug-tracking database to monitor project progress and decide about product release. Statistics are varied, but include the types of errors detected, the rate of finding and removing errors, and detection of error-prone parts of the product.

Such statistical information, however, loses its usefulness if errors and their handling are reported accurately, but passive corruption of this database takes place such as these many forms:

- Developers remove error reports claiming that they are not *really* errors, giving specious reasons like "that particular build wasn't done correctly," which is another interesting fact in and of itself.
- Managers remove error reports claiming, "We are not supposed to be testing that yet," which gives their managers an overly optimistic sense of progress.
- Testers file incomplete error reports, omitting such valuable information as the original cause of an error, which would help management detect those areas that need additional support or training.

The CMS obtains its value by making information available to all who might need it, while protecting information from corruption by those who have no authority to change it. Because of its protection function, the CMS must have the ability to restrict access. When that restriction is applied to *reading* the information, however, the whole purpose of the CMS is undermined.

In complex projects, you never know who needs to know what. For a development organization to be successful, information must flow freely. But managers may become territorial and say that certain data "is relevant to our group only." They may request that the CMS restrict access to these data, and upper management may mistakenly support them.

Why would upper management make such a grave mistake? Perhaps they fear that morale would drop if people knew the true state of the group's work. Perhaps they are trying to protect the group manager from blame. Maintaining morale and keeping a blameless atmosphere are laudable goals, but if these goals can be reached only by dismantling the information infrastructure, the project is a lost cause.

How do you protect your CMS? First of all, understand that your CMS is not just some technician's tool, but a management tool that underlies all communication and control. It belongs to you so manage it, which means the CMS group should report to upper management, not to project management. Second, set and enforce a policy of complete and open information at all times and resist plausible sounding arguments for hiding information that is in the CMS. Third, manage your people well without blaming, because blaming leads to the desire to hide information from management [5].

Hiding Information

Projects certainly generate swarms of data, and sometimes managers argue for hiding that much data to protect workers from excess complexity. These managers restrict people's access to the CMS *for their own good*, but the side effects are ruinous. More astute managers control complexity by creating special functions to manage the data, extracting useful information in condensed form.

"... set and enforce a policy of complete and open information at all times and resist plausible sounding arguments for hiding information that is in the CMS."

Typical of such functions are technical reviews, project management reviews, and quality assurance. Because their job is to extract relevant information from mountains of data, anything that hides data from these functions also hides information from management.

Technical Reviews

Technical reviews extract relevant data by transforming technical detail into a non-technical answer to the question: "Does this work product do what it is supposed to do?" Moreover, if the answer is *no*, the technical review provides information about what else needs to be done, and what issues need to be resolved.

In other words, technical reviews are a form of testing, with three major advantages:

- They can be applied to any work product, not just code.
- They can be used much earlier in a project to save dead ends.
- They can find types of errors that may not be found by testing.

Without technical reviews, managers are at the mercy of individual reports from their technical staff. Unfortunately, this is an ineffective source, for software developers are notoriously unable to give accurate assessments of their own work. By the time (during testing) a manager discovers that a developer was overly optimistic, most of the damage has been done and the costs and time are not recoverable.

Many developers (and testers and documenters) would rather not have management know the true status of their work. They figure that "if I just have a little more time, and nobody bothers me, I'll get it all right." And, sometimes, they are correct. Unfortunately, a large project that relies on such self-assessments will *always* fail because some of these predictions will invariably be wrong, and the managers will not know which ones.

Project managers can be victims of the same optimism, but technical reviews will soon reveal the true state of their project to their own managers, unless they can somehow conceal the results. They may try to convince their managers that "technical reviews are not really needed for this particular part." They may argue that the product is too complex, or that it is too simple. They may argue that reviews would slow things down, or that their programmers are very good so nothing important could go wrong. Or perhaps they argue that the programmers would be upset to have their work reviewed (and of course we must not upset our developers, even if it costs us our project).

If project managers fail to convince their managers not to hold technical reviews of a product candidate, they may hold "reviews" that do not follow an effective discipline. Or, they may hold effective reviews, then fail to follow through on addressing the issues raised. When the review concerns a requirements document, the project managers may order the developers to make the revisions but continue designing and coding from the old documents. Then the code and the requirements become so misaligned that the requirements cannot be used for designing test cases against the code. When bogus *reviews* are done for appearances, not for impacting the quality of the product, they are, as claimed, a waste of time and destroy morale. They also prejudice the organiza-

tion against future technical reviews.

You can spot and prevent bogus reviews principally through the institution of a corps of professional review leaders trained and experienced as facilitators of the human processes involved [6]. As your organization stabilizes, you can monitor reviews with appropriate measurements that will make bogus reviews stand out from real ones [7].

Project Management Reviews

It is quite natural for complex projects to drift off course. In order to steer them back on course, managers conduct regular project management reviews. These reviews transform masses of data about project status into information that can be used by upper management to assess the true state of projects and the rate of progress. Without accurate information high-level managers cannot design actions that will bring projects back on target.

Project managers, however, often view these project reviews as unnecessary interference with their authority. Privately they will say, "I just want to do my project, without interference from above." To them, well-run project management reviews threaten to expose their imperfections, unless they can somehow manipulate the reviews to hide information, rather than reveal it. They may carefully script the reviews and rehearse them so that none of the really important information leaks through to their managers.

These misguided project managers will hide information behind a slick presentation laden with lots of irrelevant data expressed in *techno babble*. If forced to discuss risks, they will do whatever is necessary to pooh-pooh them – a position that is easier to support when upper management responds to honest risk reporting by emphasizing that risks are *not* acceptable.

To avoid this trap, watch out for reviews that are run too smoothly. Insist that your reviews use only documents and files used in actual day-to-day work, not those specially prepared for the meetings. Between meetings, spot check to see if what you are seeing are actual work products. Above all, monitor and compare predicted and actual accomplishments, where accomplishments are strictly tested/reviewed work products and not abstractions such as *45 percent complete*, and are not chunks too huge to see work products from one review to the next.

Quality Assurance

The quality assurance function transforms data into useful management information by helping to establish processes and standards that, if followed, will assure quality, and then by assuring that these processes and standards are actually being followed. By observing what people are actually doing, quality assurance can provide early warning of likely missed targets; even before product candidates are available for review or testing.

Obviously, quality assurance cannot do its job if it cannot observe what people are actually doing. One of the principal ways of hiding information is to exclude quality assurance people from various working meetings. And, if those meetings produce minutes, quality assurance people are excluded from the distribution list.

Prevent these abuses by having quality assurance report to the highest levels of management, and not to project management. Insist that minutes of all meetings go in the CMS so you can check that such minutes are available to quality assurance for every meeting. And, if people start asking you to exclude quality assurance from meetings, that is the time to dig under the rock to see what is hiding.

"The tester's job is to reveal missed targets; management's job is to protect them from being abused for doing their job ... train them [testers] in accurate, nonjudgmental communication, so you can trust what they tell you."

Degrading the Believability of Information

Those managers and developers who want to hide information on the true state of their projects see quality assurance people as an impediment. Because the quality assurers' job definition requires that they be allowed to observe *anything*, simply excluding them from meetings and minutes may prove difficult. A more *effective* tactic may be to discredit the quality assurers by saying they are disruptive, do not know enough to understand what is being done, are not team players, or are too negative. That way, reports from quality assurance can be ignored, and eventually the assurers can be excluded altogether from meetings and access to information. If the assurers then try to object to any practice, they can be further discredited by saying, "How would they know? They have not participated in anything, and they have not seen the real data."

Of course, this style of defamation can be used on anyone who speaks up: a tester, an architect, a consultant, or a manager from another area. "What could they possibly know that we, the builders, do not already know?" Once upper management believes this falsehood, any report or recommendation from such an *outside* source can be safely ignored until it simply disappears. Nobody will know the true state of the project until it is far too late to put it back on track.

Do not be put off by arguments that the quality assurance people are disruptive; if necessary, simply instruct them to observe and report, and not try to say anything in the meetings. Accusations such as this will tend to disappear if you provide skilled professional facilitators for troublesome meetings. At the very least, you will be able to believe their version of what is actually happening in meetings you cannot attend.

Testing

Testing is the best place to illustrate the degradation of believability, because testing gives the most solid information about how bad a product really is. If developers cannot discredit test results, then all their mistakes are exposed to management view.

The most fundamental tactic for discrediting testers and the information they provide is to blame them for carrying the following messages:

- "Testers are always negative; don't they have something positive to say?"
- "If they were team players, they wouldn't focus so much on what's wrong, but would help make things look good."
- "Why don't they try to understand why those are not really errors? They're not developers, so they should listen to us."

Managers who fall for these ridiculous arguments succeed in cutting off their most reliable (albeit late) source of real information about hitting targets. The tester's job is to reveal missed targets; management's job is to protect them from being abused for doing their job. So, protect your testers; definitely do not consider them some lower form of

employee, like *developer's little helpers*. If necessary, train them in accurate, nonjudgmental communication, so you can trust what they tell you.

Inserting Misleading Information

In war, enemies often conduct *misinformation* campaigns, inserting incorrect information in order to give a false impression of the true battle situation. In war, this is done intentionally, but in software projects, the misinformation does not have to be intentional to destroy a project.

For example, when issues are raised in testing or technical reviewing, reports to management may show that these issues have been assigned to individuals or groups who are never actually given the assignments. Or, when issues are classified according to severity, classifiers may be pressured to downgrade each issue "so we do not alarm management."

Demonstrations

The classical case of misleading information is the demonstration. Demonstrations are not really part of the information infrastructure because managers have known for a long time that they are almost always rigged to make a product look much better than it actually is [8]. This may be great for sales, but woe to any manager who believes a demonstration instead of data from reviews and testing.

Demonstrations may be rigged in numerous ways, including the following:

- Developers may add to the test system to support a demonstration, and then remove items because they were not really ready for testing, let alone actual use.
- A developer may run the demonstration to carefully avoid any feature or combination of features and data that do not work properly.
- Developers may emphasize showy features for their *gee-whiz* effect, ignoring essential features that are mundane, but difficult to implement.
- Developers may avoid stressing the demonstration by bypassing attributes such as security, performance, and error-recovery. Instead, they show a few *normal* activities done under the easiest of conditions.

Never be fooled by a demonstration because they are *not* product demonstrations but only a sales technique. If you want a *real demonstration*, take the product out of the hands of development and put it in the hands of an acceptance test team.

Risk Management Process

Testing, though a solid source of data on missed targets, often comes too late in a project to permit effective management action. A risk-management system attempts to identify potential trouble spots early while managers still have a chance to thwart them. However, because it is assessed early in the project, risk information lacks the solid foundation of testing and is all too easily used to mislead rather than to lead.

Any software project is replete with risks; the handling of risks is the true test of management mettle. The best managers, like the best generals, want to enter their battles with risks fully laid out in front of them and their troops. The best armies have the courage to face risks head on, as do the best development staffs. But if generals lack confidence in their troops' courage, they will be tempted to mislead their troops about the existence and seriousness of risks.

This kind of distortion places the troops in a position of always being surprised when risks are realized and ill prepared to deal with them. It also causes troops to lose faith in the wisdom of their leaders, and to believe that their leaders think poorly of them.

With an open, explicit risk management process, a manager can minimize risk distortion, optimize the handling of risks that do occur, and bolster the confidence of their staff.

Conclusions

Managing software projects, like fighting battles, is a challenging business, but it becomes well nigh impossible without high-quality information. Consequently, the first job of a successful software manager is to ensure the quality of the information needed for communication and control – to protect this information from error, loss, and distortion regardless of the source. Most of all, the general has to protect troops from blame for communicating information, regardless of how unwanted that information might be.◆

References

1. Jones, C. Software Systems Failure and Success. Boston, MA: International Thomson Computer Press, 1996: 4-5.
2. Brooks, Fred P. The Mythical Man-Month. Reading, MA: Addison-Wesley, 1982: 153.
3. Weinberg, G. M. Quality Software Management: Volume 1 Systems Thinking. New York: Dorset House, 1991: 62.
4. Weinberg, G. M. Quality Software Management: Volume 2 First-Order Measurement. New York: Dorset House, 1992: 295.
5. McLendon, J., and G. M. Weinberg. "The Blaming Organization." IEEE Software 1998.
6. Freedman, D. P., and G. M. Weinberg. Handbook of Walk-Throughs, Inspections, and Technical Reviews. 3rd ed. New York: Dorset House Publishing, 1990.
7. Humphrey, Watts. S. Managing the Software Process. Reading, MA: Addison-Wesley, 1989.
8. Weinberg, G. M. "How to Automate Demonstrations." Datamation Vol. 8 (1962): 40-42.

About the Author

Gerald M. Weinberg, Ph.D., is a principal in the consulting and training firm Weinberg & Weinberg. For more than 45 years, he has worked on transforming software organizations. Weinberg is author and co-author of more than 40 books, including "The Psychology of Computer Programming," and "An Introduction to General Systems Thinking." His books cover all phases of the software life cycle, including "Exploring Requirements," "Rethinking Systems Analysis and Design," "The Handbook of Walkthroughs," "Inspections and Technical Reviews," "General Principles of System Design," and "The Roundtable on Project Management." His books on leadership include "Becoming a Technical Leader," "The Secrets of Consulting," "More Secrets of Consulting: The Consultant's Tool Kit," "The Roundtable on Technical Leadership," and "Quality Software Management," a four-volume series. Weinberg is also known for his conferences for software leaders, including the "Amplifying Your Effectiveness Conference."

Weinberg & Weinberg
10131 Coors Road N.W.
Suite I-2
Albuquerque, NM 87114
Phone: (505) 897-9707
E-mail: hardpretzel@earthlink.net