FIREFIGHTING BY KNOWLEDGE WORKERS

Roger E. Bohn, University of California, San Diego

Ramchandran Jaikumar

Report 2000-03 March, 2000

The Information Storage Industry Center

Graduate School of International Relations and Pacific Studies
University of California
9500 Gilman Drive
La Jolla, CA 92093-0519
http://www-irps.ucsd.edu/~sloan/

Copyright ©2000 University of California

University of California, San Diego

Funding for the Information Storage Industry Center is provided by the Alfred P. Sloan Foundation

Forthcoming in the Harvard Business Review

Firefighting by Knowledge Workers

Roger Bohn and Ramchandran Jaikumar

Abstract

This paper examines what happens when engineers and other problem solvers try to do too much. We define a firefighting syndrome, which includes frequent task switching, superficial problem solving, and recurrent problems. We show that this syndrome can be explained as a reaction to managerial pressure caused by a backlog of work, and it is self-amplifying. Although firefighting cannot be completely avoided in most organizations, it can be controlled and prevented from getting worse by a number of methods, some of them counterintuitive.

Keywords: firefighting, problem solving, product development

Table of Contents

Introduction ____ *1*

The Firefighting Syndrome ____ *2*

Examples ____ *6*

Explaining Firefighting with a Simple Model ____ *9*

Counterproductive Problem Solving ____ *16*

How to Stop Fighting Fires and Solve Problems ____ *24*

Tactical changes ____ **27**

Strategic Solutions ____ **32**

Cultural solutions ____ **35**

Building a problem solving organization ____ *36*

SIDEBARS

Knowledge work as solving a sequence of problems ____ *44*

Product development ____ **44**

Marketing: ____ **44**

Invention ____ **45**

Manufacturing ____ **45**

General management ____ **45**

Internet Consulting ____ **45**

Rational rules that are wrong ____ *46*

How to select problems: ____ **46**

How to solve problems ____ **46**

Computer thrashing ____ *47*

Firefighting into Mars ____ *48*

Bibliography ____ *50*

Firefighting by Knowledge Workers

Roger Bohn and Ramchandran Jaikumar

Introduction

Invariably in technical companies there are more problems and opportunities than time or people to deal with them. There are more things that the organization *should* work on than it *can*, realistically, work on. At best, this leads to situations where minor problems get ignored. But too often, it leads to a syndrome called "firefighting".

In the firefighting syndrome, engineers, managers, and other knowledge workers rush from task to task, not completing one before another interrupts them. Things that are merely "important" but not "urgent", such as long-range development, are continually interrupted by the latest fire. Although the most urgent tasks do receive attention, productivity suffers. Some jobs get "overtaken by events" and never completed. And although a task may appear to be completed, it may need to be redone later, perhaps in another form, because it was not truly solved the first time. Management is a constant juggling act, of deciding where to allocate overworked people, and which incipient crisis to ignore for now.

The firefighting image is of rushing from one brushfire to another. As soon as one seems to be under control, the firefighters leave it and rush to the next. Meanwhile, old fires that were supposedly put out continue to

---

This research was supported by the Division of Research, Harvard Business School, and the Information Storage Industry Center at UCSD, funded by the Alfred P. Sloan Foundation. Thanks to many executives and engineers who shared their thoughts and experiences with us. The authors jointly developed the main ideas of this paper, before Prof. Jaikumar's death in 1998.

smolder, and may again burst into flames and spread. Real fire-fighters have learned they must do the mundane work of checking for hot spots and monitoring fire sites to ensure they don't reignite. And they work on long-term changes to prevent fires, and to lessen their severity when they do occur. But in the more confusing and multidimensional world of modern technical companies, the chaotic behavior referred to as firefighting persists, despite resolutions to avoid it.

This article defines and explains firefighting as practiced by knowledge workers in technical organizations such as product development groups and high-tech factories. We show how it comes about as a result of good intentions gone awry, and how, once established, it is very difficult to eradicate. We show that the potential for firefighting is inherent in most organizations that continually deals with the new and unexpected, and must therefore solve technical and managerial problems in order to compete successfully. We show how to tell if you are in firefighting, and how to get out of it. Our basic conclusion is that although escaping from firefighting requires painful actions, letting it continue is a severe drag on the effectiveness and morale of your organization.

The Firefighting Syndrome

Firefighting consumes resources and saps morale. In some organizations, it seems pervasive. Yet it has been little studied. An observer of a software engineering team in the 1990s found pervasive firefighting. One engineer described their boss as "he has been very successful, but he drives his people wild -- his operating style is that he always wants things at the last

minute, and we have no choice but to do what he wants when he wants it." (Perlow 1999) The group followed a standard pattern: "Action was delayed (on problems and projects) until managers defined a situation as an emergency, but, by then, solving the problem often involved an unnecessarily large amount of effort... if it was still possible..." And, being visible to managers in that organization was synonymous with personally solving crises, rather than preventing them or assisting others. (page 67)[1]

Back in 1981 Robert Hayes hypothesized that one reason American factories were more chaotic than Japanese factories was cultural. He wrote

> ...American managers actually ***enjoy*** crises; they often get their greatest personal satisfaction, the most recognition, and their biggest rewards, from solving crises. Crises are part of what makes work fun. To Japanese managers, however, a crisis is evidence of failure. (Hayes 1981, p. 60)

Although there is indeed a cultural component to the propensity for firefighting, we sought to understand it in logical terms. We sought answers to questions such as "Is firefighting always counter-productive?" and "Does it have comprehensible causes, or is it just a consequence of managerial foolishness?"

<<<Sidebar 1 approx. here: Knowledge work as solving a sequence of problems.>>>

---

[1] Perlow did not use the term firefighting to describe what she observed. Rather, she explained it in terms how engineers used their work time, especially the conflict

A formal definition of firefighting is elusive, and it turns out it can best be characterized as a syndrome. It has the following elements, which we will show are linked.

- ***Too many problems, not enough time*** to solve them all. There are more problems than the problem-solvers (engineers, managers, or other knowledge workers.) can deal with properly, in addition to their routine work. In this article, we use "problems" broadly, to include opportunities for improvement, development work, and a broad range of other knowledge work. (See sidebar)
- ***Incomplete solutions.*** Some problems are "patched" rather than solved. That is, the superficial effects are dealt with, but the underlying cause is not fixed.
- ***Recurring and cascading problems.*** Badly solved problems reemerge, perhaps in a different form. And poor or incomplete solutions create new problems, perhaps elsewhere in the organization. Although these may have no visible connection to the original problem, they nonetheless stem from how it was dealt with.
- ***Urgency supersedes importance.*** Long-range activities such as developing new products or developing better understanding of fundamental issues are interrupted or deferred to the point that many such projects are either not completed or take a very long time.

---

between individual work, and interacting with other engineers. Nonetheless, in our terms her research is an excellent description of firefighting in one setting.

Problems are left hanging until just before a deadline. Then they are a crisis, and dealt with incompletely.

- ***Preemption*** of one problem solving effort by another, even more urgent one. Postponing work on a task until it becomes a crisis, then dropping less urgent activities.
- ***Performance drop.*** The organization's output of products and services suffers cost, quality, and speed degradations due to the many problems inadequately solved and many opportunities foregone.

We consider any organization that exhibits three of more of these elements to be doing firefighting, and if they are chronic it has the firefighting syndrome. Any organization that is dealing with cutting edge technology will experience some of these elements. After all, there is never enough time to do everything that in an ideal world "should" be done. But if the response to these situations escalates such that other parts of the syndrome occur, we can say that the organization has fallen into firefighting.

In addition to the defining elements above, there are some symptomatic behaviors that are often seen in the presence of firefighting, and that make it worse. Selection of which problems to work on may take considerable time in meetings, and be rather *ad hoc.* Diagnosing and solving problems is done by "quick and dirty" methods rather than careful and disciplined observation, experimentation, analysis, and implementation. And

firefighting can spread from one part of the organization to another, as people are pulled from areas that are not in crisis, and sent in to fight fires.

Is firefighting necessarily bad? Clearly it reduces performance below a theoretical "ideal", but whether it is bad depends on its extent and what we are comparing it against. As we will show, some rigidly bureaucratic rules can avoid firefighting, but at the price of solving almost no problems. This would be even worse than moderate firefighting. Also, there will be times, such as launches of major new product lines or startups of major initiatives, when even a well-managed organization can get into firefighting mode temporarily. So moderate and temporary firefighting behavior is not necessarily disastrous, provided it does not become chronic. But as we will show, one of the inherent properties of firefighting is that the more intense it becomes, the more difficult it is to escape from.

Examples

Product development organizations commonly exhibit firefighting. In each project, there is always more that can be done to improve performance, cost, or the "ilities" like maintainability, reliability, and expandability. As a deadline approaches, there is always more to be done. But the biggest problems come up when one organization is working on multiple development projects. As the deadline for one project approaches and problems come up, there is usually pressure to temporarily pull people from other projects onto the one with the nearest deadline. This sets off the firefighting cycle, as we will see.

A well-publicized and expensive example of firefighting occurred at the end of 1999, when two much-anticipated spacecraft to Mars vanished. The

cause of the Mars Polar Lander's crash may never be known, but the failure of the Mars Climate Orbiter was traced to miscommunication between the Jet Propulsion Laboratory, which ran the mission, and its subcontractor. The subcontractor had used English units of measurement in an obscure subroutine, while the JPL specified metric units. This led to a navigation error which left the spacecraft about 170 kilometers low when it started its crucial aerobraking maneuver, so it burned up in Mars' atmosphere. Although the initial mistake was clearly careless, equally significant is that the relevant subroutines had never been jointly tested, and that discrepancies noted during the flight were never followed up due to overwork. (Stephenson 1999)

A NASA report about JPL and its subcontractors, written shortly ***before*** these crashes, documented a pattern of firefighting. One of the features of interplanetary spacecraft is that "delivery deadlines" are inexorably set by Newton's Laws applied to planetary motion. It is physically impossible to let a deadline slip in order to fix problems. Hence, once a project falls behind, the pressure to engage in firefighting becomes very strong. NASA's inspector general found that over five different projects, the subcontractor staff early in each project was smaller than planned. This led to delays, work-arounds and poor technical decisions in the early phases. But shortcuts and work not done early in the project led to a need for even more people later in the project. (NASA 1999) This causes further inefficiency, as the new arrivals have to be brought up to speed and coordinated by the undersized initial team, just as in Brooks' Law that "Adding more people to a late [software] project makes it later".[2] People

[2] Fred Brooks in his classic book ***The Mythical Man-Month*** observed that when new developers are brought into a software team, the original people have to take the

also worked 70 hour weeks to meet delivery dates, causing errors and long run declines in effectiveness.

A second situation where we commonly see firefighting is in the transfer from development to manufacturing ramp-up of new products. At the end of development there are many "loose ends" to take care of. In addition, problems not dealt with earlier will be found once the product goes into volume manufacturing, and as it reaches the customer. These may require redesign of the process, or even worse of the product. Coordination between development and manufacturing facilities, and cultures, makes problem solving harder. If there are too many bugs for the engineers to deal with immediately, the ramp-up falls behind schedule, yields are low, costs go up, and customers lose interest when they can't get the product. The results can be financially disastrous. Maxtor Corporation, one of the top five hard disk drive companies, lost money for seven quarters in a row due to persistent problems with transfer and ramp-ups. In 1996 it was forced to sell out to Hyundai. Ability to reach high yields and high volumes quickly has been a significant issue in the competition between Intel and AMD in the microprocessor market. Only when AMD managed to ramp its latest product quickly was it able to take significant market share from Intel in the low end of the market.

Firefighting can be found in diverse of industries and functions. The sidebar on problem solving listed some. A very different example is medical systems, especially taxpayer funded systems such as in many

---

time to tell them what has been done and coordinate future work. The net contribution from the new people can therefore be negative, especially if they are brought in near the end of the project.

European countries. These systems usually have inadequate resources relative to demand, and very long waiting times for non-emergency situations. In some, there is so much time pressure that doctors take only a few minutes with all but the most urgent cases. As a result, people get sicker, become urgent, and finally get seen. However at this point it takes more work to cure them. Thus we have most of the attributes of firefighting, including too many problems to solve, scheduling based on urgency, recurrence of problems, and worsened performance (poorer health).

We will show how to avoid or get out of firefighting in these situations near the end of this article.

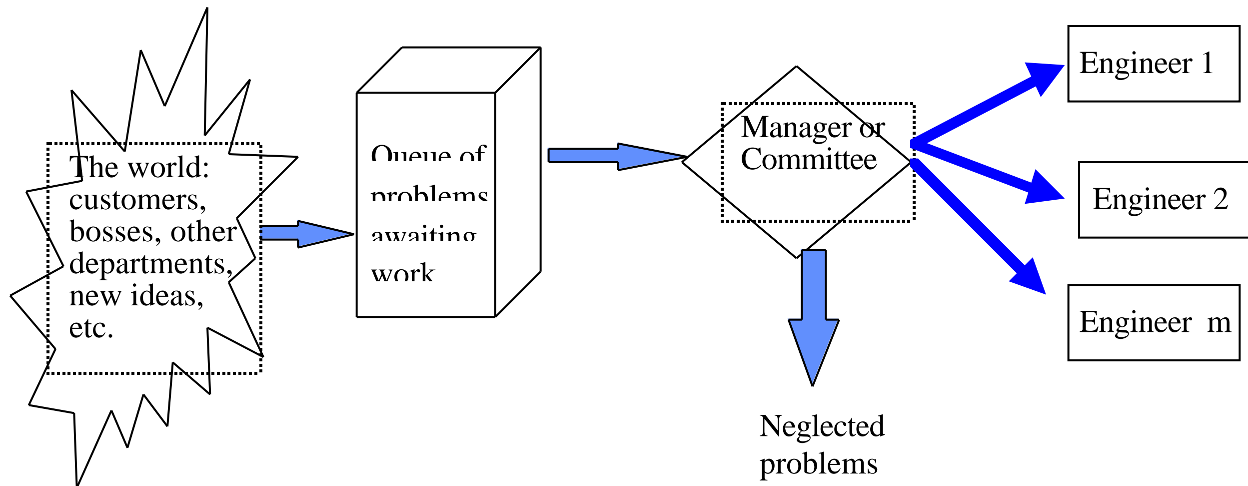
Explaining Firefighting with a Simple Model

How do all the elements of the firefighting syndrome relate? A simple model captures the essential issues. How fast and how well problems (fires) are dealt with determines whether the firm sinks into the morass of firefighting, or strides confidently forward. For concreteness we will speak of the employees who do problem solving as "engineers," although the model fits diverse knowledge workers.

Figure 1 gives a model of a problem solving organization, such as a design group, divisional headquarters, or factory engineering group. Problems have various sources, such as customer complaints, special orders, unmet design goals, quality problems, factory and supplier difficulties, edicts from higher in the organization, and even personnel issues such as the need to interview prospective group members. As they arise, problems are sent into a waiting list -- a queue --until an engineer can work on them. For

now we will assume that all the organization's problems are similar, and go into a single queue.

Figure 1 about here. **A simple model of problems and how they are dealt with.**



A manager who presides over the queue decides which problems are the most urgent and who should solve each one. As engineers finish one problem, they report to the manager who assigns the next one. Solving problems takes time to study the symptoms, confirm that the problem is real, research background information ("Have we had similar problems before? What did we do?"), diagnose the problem's causes, search for solutions, and implement the solution. Problems come in different shapes and sizes, hence need different amounts of time to solve.[3] The managers themselves may not know as much about the problems and their

[3] Obviously, real situations for allocating tasks to engineers are more complex. Each engineer may have several going at once; they may function in teams, and the teams can be different for each problem, etc. Each engineer will be better at solving some kinds of problems than others. Vacations and other "routine tasks" complicate the

importance as the engineers, so groups often help the manager decide which problems should be worked on next, and by whom.

A key number in this system is the ***traffic intensity.*** This is how busy the engineers would be "on average", or how much work they ***should*** do divided by how much they ***can*** do. It gets higher, the longer problems take to solve on average, and the more problems flow in. Adding more engineers reduces it.

$$\text{Traffic intensity} = \frac{(\text{Days to solve}) \times (\text{Number of new problems per day})}{\text{Number of engineers}}$$

As long as the traffic intensity is below about eighty percent, this system works well. As we know from queuing theory, though, when the traffic intensity gets close to 100 percent, the waiting time of problems begins to shoot up. Waiting time is the time between the problem being discovered, or entering the system, and an engineer starting to solve it. So the usual advice about queuing systems is to staff them liberally, so that traffic intensity remains below some threshold, usually ninety percent or lower.

In firefighting, however, the traffic intensity is ***greater*** than 1; problems are arriving faster than they can be solved, even if all the engineers work flat out. When the organization has more problems arriving than it has people to solve them in the time available, the queue will start to lengthen. The rate at which it grows is (at least) as fast as the mean time to solve problems, minus the mean time between arrival of new problems. For example, a testing lab in an R&D facility may get prototypes to test from

---

scheduling problem considerably. Each of these complexities reduces the attainable

many different project teams. Each prototype takes some engineering time to set up and to analyze the results. If teams are sending too many prototypes to be tested at nearly the same time, the waiting line gets longer and longer. If 8 problems arrive each week, but the existing five engineers can only handle 5 per week on average, then each week the queue of tests will get longer by 3.

Whether this leads to firefighting, or just a period of working flat out, depends on how the engineers and their manager respond. The firefighting phenomenon comes from behavioral responses to the lengthening backlog of unsolved problems in the queue. As the queue grows, in many cases to weeks or longer, various pressures grow on the engineers and their manager. Until the problem is solved, it hurts performance. And if it comes from outside or higher up in the organization, there is probably additional pressure to get it solved.

The key that sets off a vicious circle of firefighting is when various behavioral responses to the long queue lead to reduced efficiency in solving problems. Problem solving is not a fully rational and "scientific" process. Nor is selecting what to solve. Rather, all kinds of activities can occur which appear to be rational in the short run, but in fact make the problems worse. This sets off a vicious circle.

We have commonly observed the following difficulties taking place as a result of long queues of problems:

---

efficiency of the system, and makes the manager's decisions harder.

- Old problems are harder to solve than fresh ones. Memories fade, so nuances of what happened and what should be done to investigate it are lost. Conditions on a line change, necessitating the collection of new data. And many manufacturing problems are transient, so by the time an engineer investigates, the problem conditions have gone away temporarily. Many problems come and go depending on weather, different batches of raw materials, machine maintenance, and so on. To counter this, one hard disk drive company set a goal of diagnosing (but not solving) all problems on the same shift they occurred. This required 24 hour a day engineering staffs, and would not have been possible in a firefighting organization.
- Preemption -- interrupting work on one problem before it is solved to work on something else. This reduces efficiency because it takes time for people to reacquaint themselves with their previous work. New people may be assigned if the old ones are now busy with something more important - requiring even greater effort to resume work. In severe firefighting, many problems that are started on are preempted more than once, and some are never completed.
- "Overtaken by events." After considerable effort has been invested in a problem, the solution is no longer needed. The product may be slated for discontinuance, the whole development project may have been killed, or the person who requested the solution may have found a solution on their own. This is not bad *if* no effort has gone into solving the problem, but too often major effort has already been expended.
- "Maintenance" activities of problems in the queue - spending engineering time on them without actually moving forward towards a solution. Examples include responding to requests for updates about the

problem's status, and adding extra processes to temporarily deal with the effects of the problem.

- Confusion. With so much going on, and so many problem solving efforts at various stages of completion simultaneously, it gets harder to keep track of what is going on, and miscommunication escalates. Some organizations spend considerable amounts of time in weekly meetings just deciding which fires are the most important.
- Politicizing of the problem selection and problem solving processes. Some organizations revert to more top-down management, for example, thus taking decision making farther away from the engineers who know what is happening. Many organizations stick blindly to simplistic rules for problem solving, that in fact make firefighting worse. Such rules include "Always work on the most urgent problem," and "The urgency of the problem is proportional to the rank of the requestor."

(Sidebar: Rational rules that are wrong approximately here**)**

The key difficulty here is a simple one. Any engineering time that does not go into a problem that ultimately gets solved correctly is, in retrospect, wasted and makes the situation worse. Even for problems that ultimately get solved, time that is spent on "rework" or inefficient problem solving methods makes the situation worse. There is an analogy here to the "Inventory is Bad" view from JIT manufacturing. Problems that are being worked on are work-in-process inventory. They decay, they become obsolete, and they have carrying costs, namely maintenance needs and the continual damage the problem is causing. Scheduling gets difficult, and expediters are needed. Converts to JIT also find that that getting rid of

WIP inventory reduces confusion, making it much easier to see what is going on and where to concentrate attention.

Firefighting results when this situation degenerates into a vicious circle. The longer the queue gets, the more time is wasted, and the longer it takes on average to solve problems. Hence the number of problems actually solved per week goes down rather than up. Since new problems continue to come in, the queue lengthens faster, exerting even more pressure on the problem solvers, who consequently become even less effective, and so on. Mathematically, as the mean time to solve a problem goes up, the traffic intensity increases farther above 1, so the queue grows faster and faster.

We are not arguing that real systems will ever be 100 percent efficient. Even in a well-run organization, some problems that are tackled turn out, for example, to be too hard or too mysterious to solve. But wasted effort tends to go up dramatically under firefighting, in response to the pressures created by the lengthening queue.

A common part of this syndrome is that when there are too many things to do, choosing the right ones becomes driven by short-term urgency rather than long term effectiveness. We refer to that in our syndrome description as "urgency supersedes importance." NASA's recent Martian spacecraft losses illustrate this.

See sidebar, Firefighting into Mars

Counterproductive Problem Solving

The simple model of Figure 1 showed how firefighting can arise from a feedback loop involving more time to solve each problem. But there is a significantly worse situation that applies to many organizations, especially those where problem solving is inherently difficult. In these cases, the pressure of a backlog of unsolved problems leads engineers to solve problems not just inefficiently, but ***badly***. As a result, each problem supposedly solved has a chance of creating a new problem, and sometimes more than one.

A common version of this phenomenon is when engineers use their "expert judgment" to come up with a solution that they "know" will work, based on a "gut feel" diagnosis. One senior manager in a semiconductor firm rationalized his use of gut feel by calling his guesses "gut facts." They then change the process on the assumption that their guess was correct, and see if the problem goes away. If it doesn't, they implement a second change to the process, often without undoing the first change. They keep this up until they can convince themselves, and their bosses, that the problem has really gone away.

What is really happening, most of the time, is that the changes they made were essentially irrelevant to the problem. When the problem appeared to go away, it was because of some invisible shift in external conditions or upstream in the process, involving the ***true*** and still unrecognized cause. But so many changes have taken place in the last month, in so many parts of the process and chasing so many different problems, that the organization decides to "leave well enough alone." The presumed solution now becomes enshrined into operating procedures. In fact all too often

changes made in one place have created new problems elsewhere. But because the confusion and process variability level is so high, these effects are not visible.

Primo Levi, the noted author, worked as a chemist in a badly damaged paint factory shortly after WW II. Many years later, he talked to an old friend who worked there after Levi. This friend told him of an anti-corrosion paint they were making that contained a compound likely to *accelerate* corrosion. When the friend had questioned management, they said the paint had always been made that way, clearly it was necessary, and not to change anything. In fact, Levi had been the one who first included this compound in the formula. He did it as a strictly temporary countermeasure to contamination in an important raw material from a supplier, but when he left the change became institutionalized. We wish that modern information systems had made this kind of misjudgment less common. But modern process complexity has increased exponentially and in parallel with the power of information technology. Furthermore, computers are good at keeping track of *information*, but they are not good at managing *knowledge*.

This kind of hasty problem solving is often called "patching". Patching actually breaks down into three kinds of bad behavior: degenerate problem solving, no validation, and superficial solutions. First is *degenerate problem solving:*, also called "trial and error". It typically consists of trying one possible solution after another, without ever determining the true cause of the problem. Incompetent auto mechanics and computer technicians, in the days before reliable computerized diagnostics, used to

do "diagnosis by replacement." They would simply swap components, one after another, until the problem seemed better. The customer paid for all the changes, valid or not.

We recently helped a semiconductor company solve a major yield problem. The company fabricated its wafers in the U.S., then assembled them into finished devices in a different facility. To reduce labor cost, the assembly was transferred to Asia as the product was coming on the market. The yields crashed, and a previously minor type of failure now destroyed half or more of the devices. Customers were screaming for more product, but the company was unable to meet its agreed output targets. The result was an outbreak of firefighting. A problem solving team was created, and a senior manager was sent to the Asian factory several times. Each member of the problem solving team came up with their own theory about what could be happening, and how to fix it by changing the Asian process. The Asian factory dutifully implemented one potential solution after another, on a trial basis. Because of production capacity constraints, all of these trials had to use small samples, too small to detect anything except major yield changes. Equally bad, the high levels of WIP in the factory meant that it took about a month to get the results back from each trial.

When we were called in, this degenerate problem solving had been going on for months, and the team was no closer to understanding the cause or causes of the problem. For example, they had not run a controlled trial where the same batch was assembled in both Asia and the U.S. Until that was done, there was no proof that the problem was due to a difference between the two ***assembly*** facilities; it could have been due to a change in ***fabrication*** that happened to coincide with the factory move. After all, the

fabrication process was ramping up at the same time, and in addition the company was opening a new fab to make this product line. Ironically, this company had a management that espoused a commitment to modern quality methods and systematic problem solving. But, once the pressure from customers got too great, it fell back on degenerate problem solving "in order to get a quick solution."

Once we got agreement to proceed systematically, we made no attempt at a solution until we understood the underlying cause of the problem. The team then needed about three months to find the cause, which turned out to be a previously unknown physical effect. Once the cause was understood, solving the problem was straightforward. In addition, based on this new knowledge the company was able to improve yields on many other current and forthcoming products. Its new knowledge gave it significant advantage as competitors grappled with similar problems.

The second form of patching is ***failing to validate and cross-check*** a solution before making it permanent. Validating requires using a controlled experiment to prove that the supposed solution works, and that removing the solution makes it come back. Cross-checking refers to looking elsewhere in the process to see that no new problems have been introduced. Computer software is notorious for situations where fixing one bug creates new bugs. Each time a process is changed without careful testing, there is the potential to cause new problems there or elsewhere. The result is propagation of problems, and more problems to work on. Because of lags and lack of careful record-keeping of changes, the fact that the new problem was caused by a deliberate upstream change is usually not visible. For example, a "low tech" metal working factory was divided

into four successive departments. An upstream coating department changed its recipe to improve its process. Months later, the product developed major problems at the customer, who then rejected many batches.

The third form of patching is ***superficial solutions***, which fix the surface causes but not the underlying causes. An impoverished roommate of one author bought a very used sports car. One day, the motor failed on the highway. Eventually the mechanic diagnosed the problem as a faulty fuel pump, which he replaced -- problem solved. Two months later, same symptoms, same cause, same solution. The third time, my friend demanded a more careful investigation. It turned out that the fuel cap had a pressure relief hole that had been painted over, creating a pressure differential that overtaxed the fuel pump, burning it out. Once the root cause was diagnosed, it took only a minute to poke a new hole. Everything before that, was a patch.

There are conditions under which superficial solutions are acceptable. For example in software a common quick fix is to add code that checks for a particular error, and if it occurs to give an error message and stop further computation. In manufacturing quality problems, a common approach is to add another inspection step, and reject or rework units where the problem occurs. This raises costs, but it avoids passing the defect downstream to the customer. Superficial solutions are acceptable if several conditions are met. First, they must ameliorate much of the damage being caused by the problem. Second, they must stay fixed, so that no later effort will have to go into redoing the problem solving. Third, they should have a better

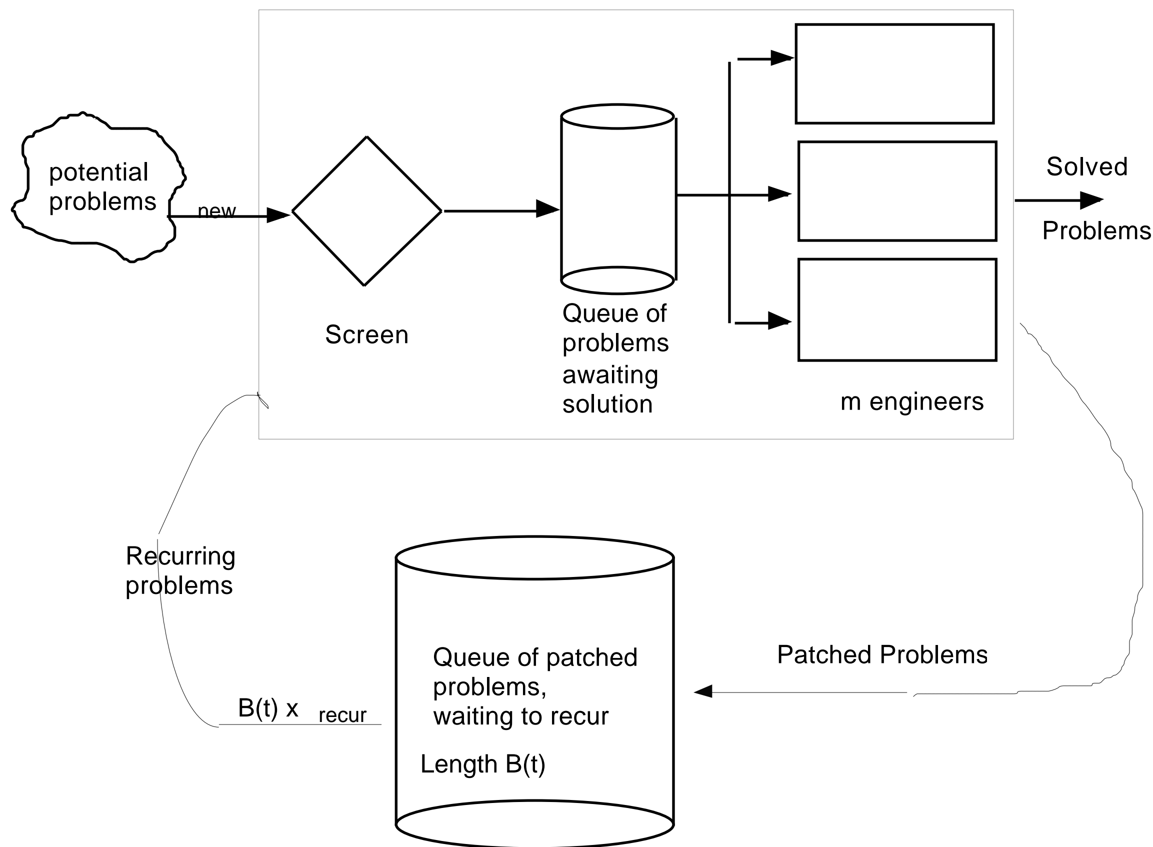
benefit/cost ratio than other possible solutions. The key ratio is (defects avoided) divided by (engineering resources needed to implement). Note that to tell whether these conditions are met, the underlying cause must be understood even if it is not fixed.

Why is patching so destructive? First of all, the problem is often not really fixed, and it will return whenever the invisible true cause returns. (Bohn 1994) Each time a problem is presumed solved, there is a chance that it is really going into a reservoir of hidden problems. If the solution was not cross-checked, it may even create more than one new problem. These lurking problems are likely to crop up again some time later. When they do, they may have different symptoms than the original occurrence, and not be recognized as recurrences. Organizational shifts also may obscure their commonality with the past; everyone involved in the first problem solving effort has gone on to other fires.

A mundane example illustrates these dangers. A steel cord manufacturer had hundreds of machines, with machine uptime a key determinant of cost and output. It therefore measured and encouraged maintenance engineers to respond to machine breakdowns as quickly as possible. But overall performance did not seem to improve. Only after careful records were kept and analyzed, machine by machine instead of person by person, did it become clear what was happening. The engineers were constantly interrupted while repairing one machine, by failures of another. They would then make a quick fix and go to the new machine. Each original machine breakdown generated from two to seven visits. On average, a problem was patched three times before finally being solved.

With patching, firefighting is driven by an even stronger vicious circle. A backlog of problems leads to hasty and counterproductive problem solving, which decreases solution rates and increases the number of hidden problems, even though it superficially speeds up the solution of each problem. As the number of latent problems grows over time, the frequency of their return grows also, until a large fraction of the incoming problems are actually old ones returning. Thus, the average time between problem arrivals gets worse as a result of firefighting, not just the average time to solve a problem. In Figure 2 each poorly solved problem may, on average, cause more than one new problem to be added to the hidden problem queue! This further accelerates the firefighting spiral.

Figure 2 about here. Problem solving, with a reservoir of hidden problems that recur



Problem solving in organizations operating in firefighting mode becomes increasingly ad hoc. In such organizations, politically successful engineers are those who believe, and can persuade their managers, that their actions eliminated the problem at hand. Often in such circumstances, cause and effect cannot be demonstrated reliably. The environment becomes increasingly chaotic and "noisy", meaning that there is much day to day variability in results with no visible causes. Rising noise makes it increasingly hard to run good experiments and diagnoses to pin down problems. (Bohn 1995)

Firefighting is likely to get worse and worse as queues grow indefinitely, potentially until the ability to solve problems collapses completely and overall performance of the factory or design group becomes unacceptable. At that point senior management faces unpleasant choices. Only drastic action may suffice, such as outsourcing a large fraction of the work, shutting down and "starting over" in some fashion, or bringing in a massive infusion of outside help. Such turnarounds are a major drain on money and management time. In essence they are fires for senior managers.

We now turn to ways to cure firefighting before the situation reaches this point, or even better to avoid it.

How to Stop Fighting Fires and Solve Problems

Given the self-perpetuating vicious circle nature of firefighting, getting out of it once you are in it requires determination. An incremental approach may not work, since it requires a large "kick" to get the backlog of work down to where the organization no longer feels pressure to firefight. Any improvement smaller than this can make things better, but probably only temporarily.

Figure 3 shows the effect of queue length (backlog of work) on the growth of the queue. It illustrates an organization where the traffic intensity without any firefighting is 90%, meaning that the engineers can do everything in 4.5 days per 5 day week. As long as the backlog of work is small, the engineers and their managers feel no pressure to take shortcuts and engage in firefighting behavior. However, if the backlog goes to the

right of this point, for whatever temporary reason, firefighting behavior begins. For low backlogs it is not very severe, and it is balanced by willingness of the engineers to work overtime. But once the backlog grows to a critical level, the amount of firefighting behavior increases to the point where the engineers cannot work any harder --additional hours are balanced by slower and lower quality work. At some critical point, shown as the "tipping point," they will start to fall farther and farther behind. Furthermore, the bigger the backlog, the more pressure, the more firefighting, and the faster the rate of growth. This is shown as increasing vertical height to the right of the tipping point.

Figure 3 approximately here (see end)

Of course, once the rate of queue growth becomes positive, the situation moves farther to the right each week. Because patched problems may not cause trouble right away, this won't all happen immediately, but eventually the past shortcuts will catch up and the speed to the right will go up. Therefore, the tipping point in Figure 3 is an ***unstable equilibrium.*** Any starting point to the left will improve, while any starting point to the right will get worse.

One-time changes in the backlog, for example the arrival of a major new product in a factory with a number of initial problems, corresponds to a jump to the right on the horizontal axis of Figure 3. An improvement in the rate of problem solving due to better methods, more engineers, or other causes, moves the whole curve down. Now we see that if the backlog gets much beyond the tipping point, a small one-time reduction in the backlog,

or a small improvement in the rate of problem solving will not be sufficient to get below the tipping point.

Now we see why firefighting tends to be common in some industries and companies. In order to maximize profit, the problem solvers are expected to work at close to their limit, i.e. only slightly below the horizontal axis in Figure 3. But the amount of work is subject to random as well as cyclical changes. Random decreases have little effect, since the backlog cannot go below zero. But random increases can easily push the organization to the right of the tipping point. Once there, it's hard to escape.

Organizations have found a number of ways to avoid or break out of firefighting. We loosely break them into three categories: tactical, strategic, and cultural. Tactical methods can be put into effect quickly, during the current projects, without high-level policy changes. Strategic methods take a deeper commitment and more time to implement. Finally, cultural changes require shifts in the culture of the whole organization, including senior management. In fact, cultural changes are ***sine qua non*** for permanently getting rid of firefighting. Otherwise, no matter how much improvement is achieved by tactical and strategic methods, a few bad financial quarters will likely lead to pressure to improve short run performance at the expense of the long run, setting off a new round of firefighting.

Remember that firefighting is a drag on performance even when the organization can keep it under control. Both the efficiency and the effectiveness of problem solving are worse, the more resources go into the

various forms of firefighting behavior. For a given number of problem solvers, they can get more done the less they have to engage in it. Therefore, even partial changes which reduce but do not eliminate firefighting can be useful.

Tactical changes

There are a number of actions a company can take immediately, that will ameliorate or in some cases solve the problem of firefighting in ***ongoing*** projects. Some of them are culturally difficult in American companies. Others are just a matter of recognizing the problem and applying resources to it.

Add temporary problem solvers. A good short-run solution in situations where the rate of problem arrivals has suddenly jumped, is to bring in temporary assistance. Urban fire fighting departments, the real kind, do this when they put out calls for progressively more firemen from neighboring areas to deal with the biggest blazes. In high tech, most hard disk drive companies have learned to send development engineers from the U.S. directly to the Asian factories whenever a new product starts in the factory. Not only do these visitors provide more trouble-shooters, they also have special expertise because they may have seen related problems during prototyping. Furthermore, this has good long term incentive effects because the American engineers know that patching problems in development will lead to more problems during ramp-up, leaving them stuck in Singapore for longer when ramp-up comes. Even individual engineers know that if ***their*** part of the design is still buggy, ***they*** will have to stay longer than their friends.

There are potential drawbacks to this solution, of course. First, it only works when the excess workload is cyclical rather than chronic. Second, if you pull knowledge workers from other parts of the organization, you risk setting off firefighting in those areas. This is what happened with the NASA Mars missions. Engineers were pulled from early stages of one mission, to work on the previous mission as it neared critical deadlines. This forced the few engineers left behind to engage in firefighting that decreased their effectiveness, causing the cycle to repeat. This can be a general issue in development groups that are working on many projects of different vintages at once. (Repenning 1998) Third, if you use knowledge workers who come from outside, they will be unfamiliar with the situation, and you must deploy them in a way that avoids Brooks' Law. For example, they can take over more routine or independent tasks that require less familiarity.

Reduce the backlog by fiat. Development companies sometimes decide to push out a release date, to give more time for problem solving and to avoid firefighting. This gives more time, to solve the same number of problems. A variant of this is to chop off a number of features from the planned release, and release them later as version 1.1. Microsoft, for example, does both with many of its releases. What became Windows 98 was released two years after the original target, and it did not have some features that were originally planned. There is a three-way tradeoff here: is it better to release a product late, with fewer listed features, or with more bugs and more firefighting prior to release? Microsoft Windows 2000 shipped with an estimated 28,000 known bugs. Microsoft Office 2000 reached the market in June 1999; nine months later the first major bug fixes came out, and ran to more than 300 bugs.

Limit the queue size: A more proactive way to reduce the queue is to prevent it from growing in the first place. When the number of open problems gets too large, shut down until they are solved. Or, only allow a new problem to go into the queue when an old one is removed. In our observation, this is something that most non-firefighting organizations do instinctively. For example, some Hewlett-Packard development centers will shut down a pilot line for the rest of the day once it reaches a certain backlog of unsolved problems. The reasoning is that until these problems are solved, there is no stable baseline for detecting and solving additional problems.

Maxtor Corporation, the same company that sold out to Hyundai because of persistent problems with new product ramp-up, subsequently turned around when it developed a very deliberate new product introduction process. As part of this process, the new product is introduced to the factory first on only a single temporary "learning line," where the engineers initially outnumber the workers. Only when this line is debugged and moving smoothly are the next lines started up. Most of what was learned on the first line can be applied immediately to avoid problems on other lines. In addition, like HP Maxtor was willing to shut down the first line for a day when the number of outstanding problems grew larger than the engineers could deal with. (Terwiesch and others 1999)

In our experience, few companies have had the fortitude to limit the queue size during normal operations. One exception is the famous JIT line-stopping rule pioneered by Toyota does this. In that rule, a worker can stop the line if they see a problem they cannot fix immediately; then

problem solvers converge on it and fix it immediately. In this way, the "queue" of unsolved problems can only have zero or one member.

Triage. A more controlled way to limit the queue size is to do deliberately what will happen anyway: admit that some problems will not get solved. The triage technique, borrowed from military medicine, controls queue size by regulating entry. Rather than let problems queue up indefinitely, or work their way through the queue only to be worked on sporadically and likely either patched or dropped, make the decision whether to commit resources to a problem when it first comes up. This technique is organizationally difficult, as it is much easier to tell someone "We'll get to your problem as soon as we can," and delegate it to someone you know is overworked, than to tell them "We have looked into your problem, decided it's not critical enough, and if you want your problem solved you will have to deal with it yourself." This is tough, but if you most likely aren't going to get to their problem, at least they are freed up to think about other ways to work around the difficulty.

Too often, when triage gets done in firefighting situations, it is based on the rule that "the urgent drives out the important." Many companies end up putting work on visible opportunities to improve at the bottom of their priority list. Another popular rule is "The urgency of the problem is driven by the rank of the requestor." (see sidebar) This form of priority setting is both a symptom and an accelerant of firefighting.

Insert sidebar about here: Rational rules that are wrong

Avoid most patching.

In the short run, patching almost always appears to be faster than solving problems systematically. Therefore, it is one of the first responses by engineers who are being pressured to go faster. Whatever they were taught about systematic problem solving goes out the window, to the cry of "It's a nice idea, but we don't have time for that stuff." A more honest explanation is "Maybe we will be sorry in the long run, but by then everyone will have forgotten my role, and maybe I will have moved on anyway." But in the long run, patching fails to actually solve the problem, and it may create new ones. We believe, therefore, that in most settings it is better to ***solve a few important problems well***, than to ***patch a number of problems, hastily***. The net losses due to bad patching will outweigh any gains from lucky guesses.

This is a controversial principle. One counter argument from some senior managers is "My engineers work harder under pressure." The implication is that "harder" is the same as "better," so managers should demand ambitious results, then get out of the way and let the engineers choose their own way of working. We observe that this is true only up to a point. Once the pressure gets too high, engineers start taking shortcuts, not just working more hours.

There are a few situations where we agree that modest patching is justified. Most problems can be solved at different levels of depth. A hasty or superficial solution is both faster and less useful, than a careful or deep one. The key determinant is the benefit/cost ratio, where cost is measured not in dollars, but in hours of problem-solving time. If a problem is of very low importance, e.g. it affects only a tiny number of end users and they can figure out ways around it themselves, then it is best to invest zero

time on the problem. If the problem is a serious one, or a hasty solution could have very adverse side effects, it is best to find the root cause and solve it well. But for intermediate problems, there may be cases where the highest benefit/cost ratio is reached by doing a superficial solution.

For example, with complex software, fixing problems requires extensive checking to make sure no new problems have been created. This is true for both outright bugs, and new features not yet implemented. In that situation, a workaround can be to flag the existence of the problem, perhaps limit the damage it can do to the user, and move on. If enough users later complain, the problem can be bumped up to a higher level of priority for the next release. For example many companies, including Microsoft, put out "release notes" describing various problems. In other cases they don't publicly list the problem, but put it in the customer support database, to be revealed to anyone who calls in with that particular symptom. Either patching approach can have a higher benefit/cost ratio than actually fixing the problem, because it can be done quickly and yet the problem only affects a few users.

Strategic Solutions

Strategic approaches to firefighting take longer to implement, but pay off across a range of projects and over long periods. Even if they are not sufficient to avoid firefighting, they will increase the number of problems that do get solved.

Change design strategies. The last decade has provided new insights into how to design products. In a number of industries, companies have had great success in increasing the commonality of designs from one generation to

the next, and across products. Not only does this reduce the number of design problems that must be solved within and across product generations, it also reduces the changes and therefore the problems in manufacturing. Commonality can be further enhanced by modular designs, which allow improvement of one section of the product without much change to the others.

For example, hard disk drive companies used to have separate teams on each successive drive generation. This led to designs where even the screws changed with each product. But as generations fell to less than 18 months, gradually the surviving companies learned how to use the "platform" concept effectively. Now, the capacity of a drive can be doubled by changing the heads, media, and firmware, and substituting the latest and fastest signal processing chips on the circuit card. The new design is manufactured almost exactly as the previous one was, and the problems that come up in the next generation are concentrated in a few areas. An example of the success of this strategy is that Seagate can now transfer some products into manufacturing with only a few development engineers having to move temporarily to the factory, whereas before 20 or more would be needed.

Outsource some parts of design: Companies in the auto industry have moved to more "black box" design, where they specify only the characteristics of a subsystem, including its size, weight, power requirements, and performance. The subcontractor who will build the subsystem then determines the best way to achieve these objectives, including what new technologies to use. While the total number of problems may not go down, they have been outsourced away from the central design team.

Develop more problem solvers. One of the successes of the TQM movement was that in some companies, non-engineers were trained to solve problems. Even though they are not as fast or as broad problems solvers as an engineer, by handling many of the more mundane problems, technicians and workers can free up resources for the more difficult ones. However, this advantage can be eaten up if they are led to merely patch problems, again and again.

Software companies have developed a variant of this method in their use of beta testers -- future customers who use the software in beta form, before it is completed. Although the beta testers cannot ***solve*** the problems, they can ***find*** them, reducing the number of engineers who need to do testing. Microsoft claims to have had 750,000 beta testers for its Windows 2000 operating system.

Better problem solving tools and methods. Better simulation tools, and increased understanding of problem solving as a task, make it possible to improve the problem solving speed and accuracy of individual knowledge workers. Higher level methods, such as structured design and programming methodologies, can also be quite effective. However, there is a potential trap here: ***the introduction of improvements can itself permanently worsen performance.*** (Repenning 1998) While new methods are being introduced, they inevitably ***reduce*** engineering productivity at first, since it takes time to learn and first use the new tools. If the introduction is too rapid, the effect can be enough to tip the organization into firefighting. See figure 3 - reduced productivity equates to raising the entire line. Once the tools have been assimilated, it may be hard to get back to the left of the tipping point,

since the accumulated firefighting will have created many new problems in the queue.

Cultural solutions

As we showed in Figure 3, the tipping point is an unstable equilibrium. Even in an organization that has implemented methods to avoid firefighting, and is normally well below the horizontal axis, there will still be occasions when extra work arrives, and pushes the backlog to the right of the tipping point. What happens then is pressure to enter firefighting. Put differently, it is almost always possible to make a temporary and apparent gain in performance firefighting: concentrating on short run payback at the expense of important problems, patching in the hopes that nothing will happen right away, and other shortcuts.

At these times, the organization's problem solving culture becomes critical. If managers are too far removed from the problems to see the consequences, and if the reward system favors firefighters, then the vicious circle will begin again and the system will tip into permanent firefighting. Avoiding this depends on the culture of middle and senior managers. We suggest the following guidelines:

Don't reward firefighting. In most American organizations, the hero is the one who puts out the most visible fires. But this leaves the questions, "Where were they when the problem started? Why didn't they intervene sooner, before it got this large?" Instead, reward managers who don't have visible fires, and who practice long-term problem solving.

Don't push to meet arbitrary deadlines at all costs. Such incentives always favor firefighting "just this once." Instead, measure development projects by how few problems remain afterwards. Most companies have measurement systems that have some count of "open issues" and problems discovered post product release. And good factories, although by no means all, have accurate lists and measures of unsolved problems. If this list stays the same or grows for more than a month after a product introduction, the factory is in firefighting mode.

Accept only validated and cross-checked solutions, not patches. We have explained the importance of this, but enforcing it requires informal checking at all levels of management. One of the advantages of Intel is that its management, from the CEO on down, has extensive line problem solving experience, and can detect the difference between a patch and a real solution. When subordinates find that hasty solutions will come back and bite them, organizationally speaking, they will stay away from them.

Building a problem solving organization

In today's highly dynamic business environments, innovation, improvement, and dealing with the unexpected are key tasks. The unexpected takes the form of problems, which, when solved, open the door to innovation and improvement.

When problem solving is *ad hoc* and unvalidated, problems frequently recur and propagate. The more problems, the less time to work on any one. Problems are dropped for want of resources, rendering the time spent on them essentially wasted. At the extreme, problems overwhelm the

organization's problem-solving resources. This is the firefighting organization. As we discussed at the beginning of this article, a number of seemingly unrelated symptoms in fact are part of the firefighting syndrome, and both cause and are caused by it.

Contrast the foregoing with the problem solving organization that triages problems, dismisses those of low priority out-of-hand at the outset, solves and validates rather than patches the high priority problems, and cultivates a problem solving environment. Although they may fall into firefighting from time to time, they quickly dig out.

We have observed two additional elements that assist such organizations. First, they have learned to go beyond solving individual problems, to solving "problem classes." Second, in manufacturing they do much of their problem solving on "learning lines".

Problem classes are a way of grouping seemingly diverse problems, determining an underlying set of issues, then learning about those issues. Once they are understood, solving the specific problems is often straightforward, including going beyond the original problem set to solve some that the organization was not initially aware of. Equally important, it usually gives deeper insight into the processes, that can be used in designing the next generation of products and processes.

A historical example of problem classes was the discovery of operator-caused particle contamination in the semiconductor industry. When integrated circuits were first manufactured there were, as there are today, a multitude of defects. Most circuits on a wafer did not work, and had to be

thrown away. Companies developed codes for describing the various failures based on which tests the circuits failed. When the losses due to a particular code were too high, a team would try to figure out why, and solve the problem.

It eventually became clear to engineers that many problems were caused by particles falling on the wafer surface and ruining the tiny circuits. Since the circuits were built up photographically, there were many different places where this could happen. "Particulate caused defects" was an example of a problem class, since it shifted the problem solving from specific failure codes, to finding sources of particles. This insight led to the redesign of manufacturing facilities to be in "clean rooms", where air filters removed particles. As circuits got smaller, ever-smaller particles became problems. Another breakthrough came when it was realized that people in the clean room were a major source of particles. Again, this was a problem class, "contamination coming from people." By dressing people in special garments, and eventually forbidding use of any makeup or other skin coatings, many contamination problems could be suppressed at once. Contamination control is still evolving today, and has become an engineering specialty in its own right. Entire firms specialize in building clean rooms, while others provide special lint-free and static-free garments.

A more recent example from the hard disk drive industry is the discovery of micro-shocks. During assembly of a hard disk drive, it is very delicate since the protective mechanisms have not yet been built. Recently, engineers began to realize that movements of only a few inches by an

operator could cause a very short duration but high acceleration shock, sufficient to damage the components. These shocks were invisible to normal techniques of measurement, since they were so brief. With this new problem class, a number of different kinds of processing damage could be tackled at once.[4]

Manufacturing is not the only place where problem classes come into play. Examples of problem classes in designing products for consumers are given by Donald Norman, in his book ***The Design of Everyday Things.***(Norman 1988) For example, look at the knobs used to control devices such as cars and appliances. Norman deduced a set of principles, such as to differentiate controls that do different things by multiple methods (shape, size, color, texture, sound when turned, etc.). Standardization and analogies also are useful to reduce learning time. The design problem then shifts from "how should we design this knob" to "how should we design this set of knobs for our target market."

The drawback of trying to use problem classes is that it requires a deliberate, extensive, and sustained commitment to formal problem solving. Among other things, it requires correlation of information collected from different parts of the organization over long periods of time, development of scientific models that yield higher levels of process understanding, and controlled experimentation in the factory. These are exactly the kinds of long-run activities that are neglected during

---

[4] Both of these examples are problem classes that became known to the industry as a whole, and to equipment makers. There are many single-company examples of problem classes, that have not yet become general knowledge. Such information is often extremely proprietary, since even knowing the name of a common class of problems can be very useful information.

firefighting. However, they are routine and easy in learning lines, which we now turn to.

Learning lines are production lines within a factory or other process, that are run to maximize detecting, learning about and solving real problems. Thus they are not pilot lines, which use special purpose equipment, special operators, and so on. Rather, learning lines are in the real factory, using standard raw materials, real machines, real operators, making real products for real customers. They are exposed to all the vicissitudes and opportunities of the rest of the factory, such as bad material batches, unreliable machines, and careless operators. In what sense, then, are they run to maximize learning and problem solving? They are the locus of data gathering, engineering trials, and observation by problem-solving teams. Typically special "off line" instruments are installed. In contrast to most computerized processes, data from the learning line is actually kept and analyzed. This is useful, among other times, when a problem is detected some time after it occurred.

The learning line facilitates problem solving because it is designed to have all the same problems as the rest of the factory, but to make them more visible and solvable. Such lines are often used for experiments with new ideas, as well as diagnostic experiments. Often they have performance which leads the rest of the factory, since problems are detected and solved more rapidly, and new innovations are put in place there before rolling them out fully. Part of the art of using a learning line is to make sure it faithfully emulates the rest of the factory, while improvements are quickly transferred from it to the rest of the factory. This is accomplished by not separating engineers on the learning line from those in the rest of the

factory. Rather, any engineer can use the learning line as his or her laboratory for investigation.

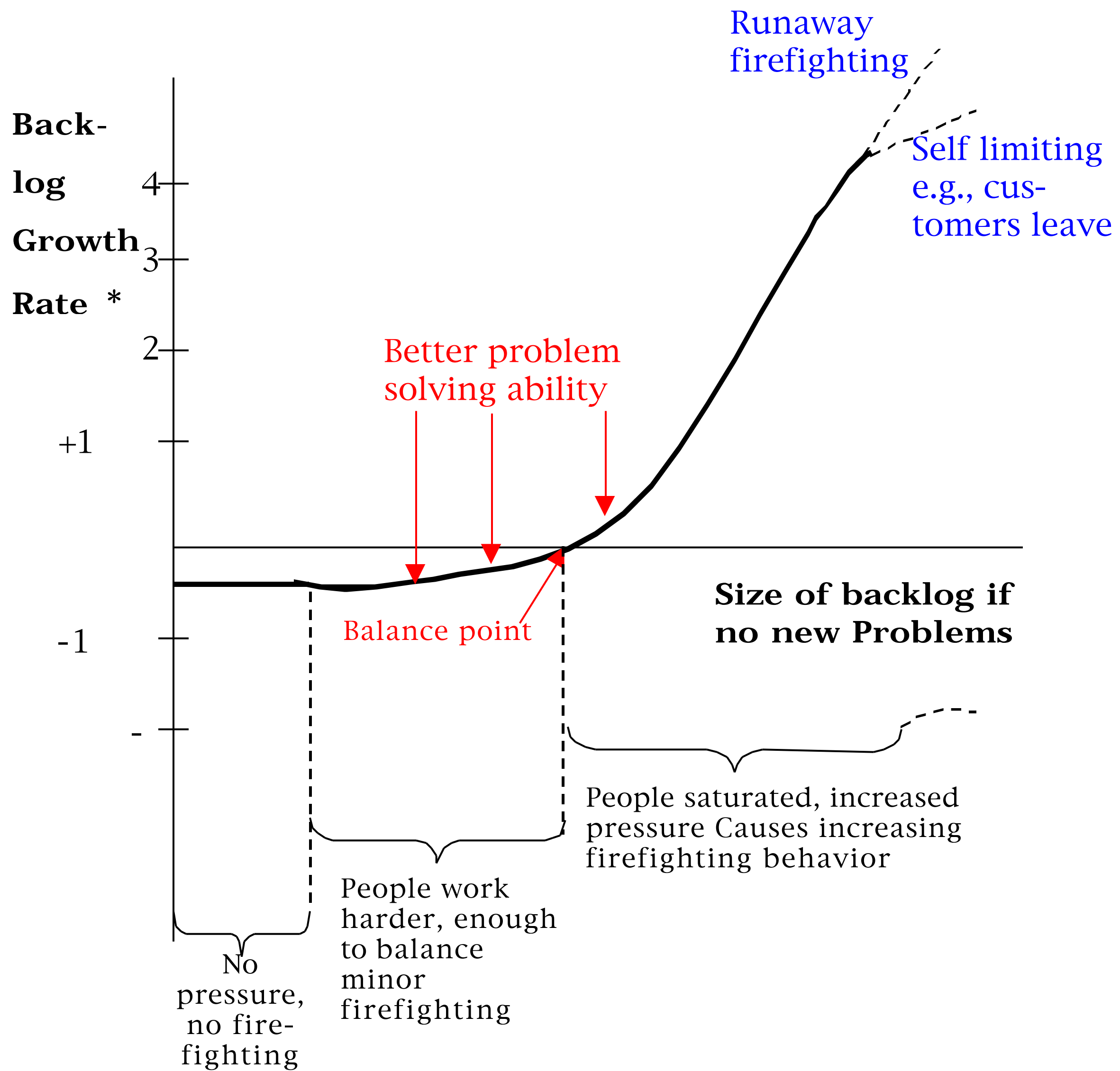
The use of learning lines is very rare in product development, but more common in marketing. There the problems being investigated are those involving customers. The learning "line" is therefore a group of special customers, who are tracked more closely. They can be used for early warning of problems with the company's products, for identifying unmet needs, and as a general source of feedback.

How does the firefighting organization transform itself into a problem solving organization? Learning lines and problem classes are just tools, not cultural shifts. First the organization must recognize that the self-perpetuating firefighting syndrome is not an inherently irrational response to high-pressure management situations. Its genesis rather is a set of managerial decision rules and behaviors that, although seemingly reasonable and effective, have the effect of causing firefighting in the long run. The problem-solving organization must not only abandon these seemingly logical practices, but also adopt techniques that at first blush can seem irrational, such as deliberately ignoring some problems.

These are not easy things to do. Established organization culture often mitigates against them. Management that believes that its workers "work harder (read better) under pressure" mitigates against them. Even short-run logic mitigates against them. What mitigates for them is the constant drain on knowledge workers that firefighting causes. No matter how hard they work, firefighting extracts a "tax", typically more than 20 percent, on their efforts. At the extreme, firefighting caries the risk of having to

withdraw a product line or shut down a plant that has been rendered utterly ineffectual by firefighting. The benefits are clear, the road difficult, the choice unambiguous.

Figure 3: Unstable equilibrium



- Rate at which backlog grows
  = Rate of arrival - Rate of solution

SIDEBARS

Knowledge work as solving a sequence of problems

Nobel-prize winner Herb Simon pointed out decades ago that managerial decision making can be viewed as solving a series of problems. Changing or improving something can be viewed as solving the problem, "How can we do this even better?" Top level problems usually require solving a chain of nested smaller problems. How well and how quickly the problems get selected and solved determine how fast the organization adapts to change and improves over time. We use problem solving in Simon's very broad sense, to include fixing things, improving, and inventing.

The problem solving framework has been exploited most heavily in product development (e.g. Clark and Fujimoto; Clark and Wheelwright), but it applies well in other areas too. Here are some examples of problems that are faced by knowledge workers in high-tech companies. It's often useful to state them as questions.

Product development

- What features are essential in the new product? Which are desirable but not essential?
- What should be the overall product architecture. How should we design this specific module?
- We can't meet our power consumption budget for this handheld device with the present design. Should we use a bigger battery, take out some functionality, or redesign it? If we redesign, what approach is likely to get power requirements down to the target with minimal delays in the schedule?
- We have a bug when the user shifts the display. Why, and how do we fix it?
- Manufacturing says that component will be expensive to make. Can we substitute or redesign it?
- Can we come up with an innovation in this product line that will be easy for us, but difficult for our competitor to respond to?

Marketing:

- How can we best position this product?
- What unmet needs does this market segment have?
- How should we respond to our competitor's price cuts?

- Why did sales fall in Latin America last month? Is it something to worry about?
- Which ad campaign should we select, and how much should we spend on it?

Invention

- It's too hard to see pictures and text together on the Internet. (Result: Mosaic, the first web browser.)
- What would we get if we apply secure military communications technology to personal communications. (CDMA for cellular phones)

Manufacturing

- We are getting more rejects than before. How can we remedy this?
- Can we improve the speed of this machine, without sacrificing quality?
- The machine broke. We need to fix it.
- The machine keeps breaking. How do we make it more reliable?
- Shorter product life-cycles and more products at once are demanding more flexibility in manufacturing. How should we respond?

General management

- How can I help my subordinates to grow and improve, especially in their strategic thinking?
- The customer's VP just called me to complain about our unreliable delivery. Is she correct that this happens frequently? Is this a crisis, that I should push for fast resolution? How can we satisfy her, without spending a lot of money? (This example illustrates the potential of senior management to inadvertently set off fire-fighting at lower levels of the organization. See sidebar, "Rational rules that are wrong.")

Internet Consulting

- Our client wants to "get on the Internet," but they don't have much experience with it. What is a good project to get them started? Strategically, how fast should they go?
- Our client's web pages take a long time to navigate by customers in X market segment. Should we set up a separate section of our site for X? How can we redesign the pages to be easier to navigate? Or, are the X users internet neophytes, who will get better on their own?

Rational rules that are wrong

We identified a number of seemingly rational rules-of-thumb that many organizations use in problem solving. When the organization is not under stress, they may indeed be good. They can also be good for individual knowledge-workers who are fostering their individual reputations. But once on the edge or in the middle of firefighting, they are pernicious for the organization as a whole.

How to select problems:

- ***Every apparent problem should be investigated. All real problems are worth solving.***
- ***Work on all the things you are asked to.***
- ***Always work on the most urgent problem.*** This is related to "nearest due date scheduling", in other words work on the soonest deliverable. This turns everything into a crisis and becomes self-fulfilling; other problems get left until they too approach a deadline or crisis. (Perlow, 1999) As we've seen, many problems can be solved in fewer total hours of effort if they are approached more calmly and carefully.
- ***If you don't have time to do something yourself, give it to a subordinate.*** Each time the problem gets passed down the line, the receiver needs time to examine the problem and decide what to do about it. It's better to kill the problem at the start, or pass it directly to whoever does have time to work on it.
- ***The urgency of the task is proportional to the rank of the person asking you to do it.*** Sometimes it is, and sometimes it's not.
- ***When you call someone lower ranking than youwith an urgent problem, they should take care of you right away.***
- ***When the phone rings, pick it up unless you are in a meeting. You can always go back to what you were doing.*** This encourages constant interruptions, and cuts into uninterrupted thinking blocks. In some organizations, e-mail has reduced this problem by allowing knowledge workers to answer questions when it's convenient for them. But there are still many cultural issues about when to interrupt someone.

How to solve problems

- ***Bring to meetings everyone who might be affected or might be able to help. Consult widely and inform widely before acting.***
- ***Give everyone short deadlines -- they will work harder. If they make the deadline, it was probably too long.*** It's true that most well motivated people will work harder under pressure (until they burn out). But they will also be forced to cut corners and set off the firefighting spiral.

- ***If you can't solve a problem completely, do the best you can. If you are too busy to fix something properly, do as much as possible.*** This quickly leads to incomplete and unvalidated solutions.
- ***Give all difficult problems to your best troubleshooter.*** This means they get overwhelmed, and in the mean time others don't get experience.
- ***Formal problem solving takes too long when you are busy; use quick and dirty methods instead.*** This is so important that we discuss it at length in the main text. This and the next point relate to the idea that problem solving is a pure art, not a science.
- ***Give your trusted people complete discretion about how they solve problems. Even junior engineers know how to solve problems; don't waste their time with non-technical training.***

Most of these rules can be good ones under some circumstances, and some of them are good under most circumstances, as long as they are applied in moderation. But under firefighting conditions, they exacerbate rather than ameliorate the syndrome.

Computer thrashing

Computer operating systems (OSs) can exhibit a behavior termed *thrashing* that is analogous to firefighting. An OS has many demands placed on it simultaneously. For example desktop PCs need to "simultaneously" monitor a network connection, update the clock,, interpret keystrokes from the user, update the display, perform calculations, drive a printer, accept data from the hard disk drive, collect garbage (don't ask), and so forth. And personal computer operating systems now allow end users to run several programs at once: playing a music CD, downloading e-mail, and typing on a word processor, for example. Yet almost all modern computers have only a single central processor, which switches among these various activities so that they appear to be going on simultaneously. Of course sometimes the computer has more to do than it can do immediately - the traffic intensity is greater than 1. The user sees this as a visible slowdown, for example a stutter in the music and a pause in the screen update.

In order for a computer to work effectively, the operating system needs to have rules for switching among tasks based on priorities and preemption. Some things are more urgent than others, such as handling communications inputs. But each "switch" itself consumes CPU time. This is analogous to the reduction in problem solving effectiveness that occurs

when an engineer switches among problems.[5] Systems designers learned early that if the rules governing task switching are not designed well, an OS can become consumed with task switching. This is called ***thrashing***.

One insight from the OS analogy is that no fixed set of priority rules is effective under all circumstances. If, for example, an OS must process many short and a few very long jobs, priorities should be set to ensure that all jobs get some time, but long jobs get more time than short jobs. If, on the other hand, all are long jobs, assigning equal time slices to each will result in none getting completed for a very long time. Insteade priorities should be set to ensure that some jobs are completed before time is allocated to the others. Managing firefighting, like setting priorities in OSs, will not be avoided by any simple rigid rules.

Firefighting into Mars

At the end of 1999, two much-anticipated spacecraft to Mars vanished when they reached the planet. The cause of the Mars Polar Lander's crash may never be known, but the failure of the Mars Climate Orbiter was traced to miscommunication between the Jet Propulsion Laboratory, which ran the mission, and its subcontractor. One had used English units of measurement, while the other was using metric units. This led to a navigation error which left the spacecraft about 70 miles low when it started its crucial orbiting maneuver, so it burned up in Mars' atmosphere.

A NASA report about JPL and its subcontractors, written shortly ***before*** these crashes, documented a pattern of what we are calling firefighting. One of the features of interplanetary spacecraft is that "delivery deadlines" are fixed by planetary orbits. It's not possible to let a deadline slip in order to fix problems. Hence, once a project falls behind, the pressure to engage in firefighting becomes very strong. NASA's inspector general found that over five different projects, the subcontractor staff early in each project was smaller than planned. This led to delays, work-arounds and poor technical decisions in the early phases. This required catch-up work later.

---

[5] Fortunately for users, computer memories are usually perfect, whether the delay is ten milliseconds or ten seconds. Unlike humans, they don't normally have to go back and redo more old work, the longer it has been since they worked on a particular task. There are exceptions to this, though, such as when another device "times out" due to a long delay. Then the computer, or the user, has to go back and re-initiate the action.

This was done by stealing staff from early phases of subsequent projects. Engineers also worked 70 hour days to meet delivery dates, causing errors in the short run and long term declines in effectiveness.

This is classic firefighting, with a twist that it propogates from one project team to the next. A backlog (long queue) of work early leads to inefficient work, and creates new problems that will have to be dealt with later. Since all problems must be solved by fixed delivery dates, this leads to pulling people from other teams and throwing them at the problems. This causes further inefficiency, as the new arrivals have to be brought up to speed and coordinated by the undersized initial team, just as in the Brooks Law that "Adding more people to a late [software] project makes it later".[6]

The firefighting syndrome was apparently set off when NASA and JPL switched from big projects to a "Faster, Better, Cheaper [and more of them]" approach to spacecraft. The workload at JPL went from one project every two years, to five launches in a six month period. More of the work was subcontracted out, so the workload on the subcontractors went up even more.

A more specific report by an outside Review Board, after the Mars Climate Orbiter's failure but before the Polar Lander failure, gave specific details of what went wrong with that project. Poor coordination among different groups, and understaffing of the critical navigation team were among the causes. Early warning signs were missed or ignored due to the pace of events. The navigation error probably would have been corrected by a contingency burn, but a decision on whether to do this burn was never made due to the crush of other work.

It is interesting that almost all the "corrective recommendations" of the report involve more work: more contingency planning, more attention to anomalies early in the project, more testing, and so on. Although laudable in isolation, these have an excellent chance of making the firefighting worse.

---

[6] Fred Brooks in his classic book ***The Mythical Man-Month*** observed that when new developers are brought into a software team, the original people have to take the time to tell them what has been done and coordinate future work. The net payout from the new people can therefore be negative.

Bibliography

Bohn, Roger E. “Measuring and managing technological knowledge.” Sloan Management Review 36 (1 1994): 61-73.

Bohn, Roger E. “Noise and Learning in Semiconductor Manufacturing.” Management Science 41 (1 1995): 31-42.

Hayes, Robert H. “Why Japanese Factories Work.” Harvard Business Review (July-August 1981): 57-65.

NASA, Office of Inspector General. JPL Management of Subcontractor Technical Performance. NASA, 1999.

Norman, Donald A. The design of everyday things. New York: Doubleday, 1988.

Perlow, Leslie A. “The time famine: Toward a sociology of work time.” Administrative Science Quarterly 44 (1 1999): 57-81.

Repenning, Nelson P. The Transition Problem in Product Development. MIT, 1998. Working Paper

Stephenson, Arthur et al. Mars Climate Orbiter Mishap Investigation Board:phase 1 report. NASA, 1999. Mission Failure Investigation

Terwiesch, Christian, Kuong Chea, and Roger Bohn. An Exploratory Study of International Product Transfer and Production Ramp-Up in the Data Storage Industry. ISIC, 1999. Report 99-02.