Process Evolution
in a Mad World

Originally published, 1994

James Bach
Satisfice, Inc.
james@satisfice.com
http://www.satisfice.com

Introduction

There is a guaranteed formula for success in software development.

1. Create the right environment.
2. Get the right people.
3. Determine the right course of action.
4. Make everyone do it.

Many books and papers have been written about different ways of executing this formula.

But, what if we can't afford the ingredients?

This paper is about what to do in cases where we are asked to accomplish an impossible task, in an impossibly short time frame.

We will review the three high level strategies of technical management, *leadership*, *risk management,* and *process control* and see how risk management provides the flexibility to be applicable even in a chaotic world (Fig. 1).

Then, we'll examine the question of process evolution and show how it can occur within a risk management and leadership framework, without the need for process control.

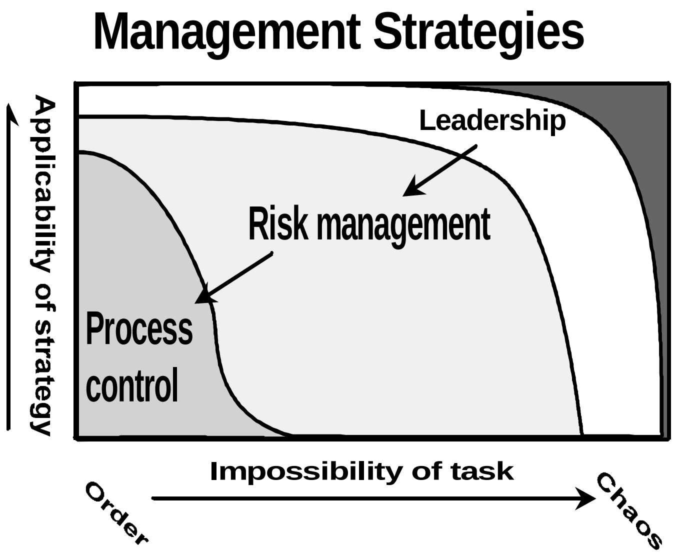

Figure 1

In order to frame the issue of management and evolution, though, we first have to examine the often chaotic nature of the world in which software projects occur, and the role of processes in responding to that world (Fig. 2).

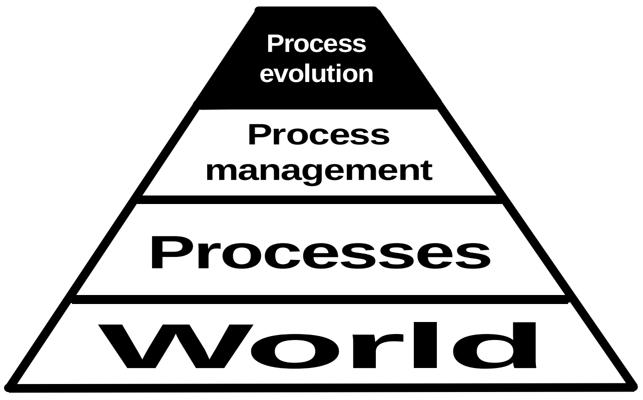

Figure 2

Order and chaos in the world

Software engineering happens within one of two worlds. For want of better terms, we'll call them the *orderly* world and the *chaotic* world. These are not meant to be dramatic terms. They refer to specific and persistent characteristics of two profoundly different dynamics of software development.

Orderly world: A working environment for software development in which problems are *visible* and *controllable*, and appropriate resources to solve problems are available.

Chaotic world: A working environment for software development in which problems are *ambiguous* and *uncontrolled*, and resources to solve problems are limited.

These worlds represent only the factors outside the direct control of a project, not the *processes* used within a project. Chaotic and orderly processes are covered later.

Most of us would rather work in an orderly world. Much of quality assurance literature is devoted to the problems of chaos and the virtues of order. Nearly every technique advanced in the literature relies in some sense upon an assumption of order.

As reasonable as that is, the unfortunate side effect of focusing on order is that we discover too little about its opposite. Does chaos have any advantages? Why can't we stamp it out? How do we succeed in spite of it?

Why does chaos persist?

QA methodologists have some of the answers. and they aren't flattering to us. Here are the most popular ones:

- We aren't rational.
- We fear change.
- We only think for the short term.
- We are poorly managed.
- We don't value quality.
- We don't understand engineering.
- We don't understand management.
- We are going out of business.

Now, many of these reasons are true to some extent in many organizations at many times. Yet, there's much more to this problem than simple incompetence or sloppiness.

Chaotic organizations have, in fact, produced excellent software. At Borland and Apple, where I have personal experience, plenty of great software has been created. And having interviewed colleagues and coworkers who at one time or another have worked at Lotus, SPC, Quarterdeck, Microsoft, Claris, and WordPerfect—all of them well-known and influential companies—it's clear that these organizations are in the same situation.

Despite very logical arguments from process pundits, something must be wrong with their premises. None of the reasons given above seem meaningful when applied to companies that compete effectively in a cut-throat world, and build huge markets out of nothing. Were the creators of Hypercard sloppy and incompetent? The Hypercard 1.0 project (which enabled widespread end-user programming that in turn created an explosion of useful applications for the Macintosh) was extremely chaotic, yet extremely successful for Apple Computer.

Since chaotic organizations can and do succeed, it indicates that methodologists who criticize them have oversimplified the nature of business, the nature of engineering and the nature of the software consumer.

Let's take a closer look at each of these three issues.

Chaos is a function of change...
Change is opportunity.

In an information society, chaos is an essential condition of business. Tom Peters' landmark analysis of the challenges of modern business, *Thriving on Chaos*, argues that general advances in technology, the proliferation of new market channels, and increasingly demanding customers have forced us into a revolution in the way business is done. This revolution reaches into every kind of business, even into the heart of engineering organizations.

The revolution is about embracing change. The problem for engineers is that change translates into chaos, especially when a single error can potentially bring down an entire system. But, change also translates into opportunity. It's as simple as this: if there is time to put a certain amount of functionality into the product easily, then there is time to put in *more* functionality at the price of a certain amount of disruption and risk. Thus does madness creep into our projects— we will tend to take on as much risk as we possibly can.

Peters identifies five key strategies to take advantage of this mad situation:

- Creating total customer responsiveness.
- Pursuing fast-paced innovation.
- Achieving flexibility by empowering people.
- Building systems for a world turned upside down.
- Learning to love change: a new view of leadership.

Taken in their truest spirit, the latter four of these strategies are the antithetical to the primary tactics of quality assurance, which are to measure and control.

Measurement and control relies on the establishment of a stable model of development, stable technologies, established processes that are detailed and specific, and people who adhere to those processes in the course of their work.

Far from adherence, Peters urges rule breaking, bureaucracy bucking, and skunkworks projects. He urges that we confront uncertainties with very proactive risk taking, rather than recoiling from those uncertainties. This obviously plays havoc with measurement and control.

But let's say that Peters is wrong. Instead, let's say that uncertainty should be met with processes that enhance order. Unfortunately, that leaves us with another large problem: getting the business to go along with us. A lot of methodologists shrug off this problem. They say, "if you don't have support from top management for quality, nothing can be done." That's the reason why so many management books are dedicated to ways we can convince management to give us the authority, staff, tools, and time we need to do a good job.

But what if that fails? What if the future of the business rides on accomplishing a particular project that takes 10 people to properly test but there is only money for 3 engineers? Do you give up?

What are the methods that help us cope with the mad world?

Projects with such constraints are everywhere. Some succeed, some fail, many succeed just enough to get by. The key is to find every possible way to optimize resources. True, the result isn't as polished as it might have been. The point is that we find ourselves in these clutch situations and we could use some tips on how to cope.

Methodologists usually make the assumption that projects can be protected from shortages, emotions, and internal or external disruptions. Methodologists assume that the world can be stopped while we settle down to do our requirements documents and DFD's, while we design, code and unit test.

That's wishful thinking.

Processes don't create software, people do. For hard problems like software development, it's important to understand that the *true* process for getting to a solution cannot be modeled. If it could be, then we could tell our problems directly to computers and they would do all the analysis, design, and coding for us. Instead, we find that only the external attributes of hard processes can be described and controlled. This leaves the rest up to a dangerously variable creature: the human engineer.

Engineers have personalities. They vary in their talents, ambitions, experiences, and world views. One engineer might be very good at creating structure charts; another might be better at pseudocode. One might be a whiz at debugging; another at defensive coding.

I once found myself in the position of being a project coordinator in QA on a project that already had a project coordinator in the development organization. At first we were concerned that his duties and mine would overlap. But, as we talked this through, we discovered that his preference was to be out and about, keeping people aware of what was going on, whereas my preference was to create databases and systems to track project information. He disliked systems; I disliked walking around. Instead of overlapping, we could work together and leverage off of each other's strength.

This was a good idea because each of us covered a weakness in the other; but the reason it actually worked was because *we liked each other*.

Methodologists assume that everyone will do the right thing regardless of their unique human strengths and frailties.

That is another kind of wishful thinking.

Requirements documents don't buy software, people do. And customers are people with all the variabilities noted above. In many cases customers are difficult to identify or analyze. We try to predict their needs, and we are often wrong. We satisfy some customers and not others. In general, for mass-marketed software, the more features we jam in, the larger the number of customers whose needs we are likely to satisfy.

Then there's the problem of competition. Competition changes the customers expectations. In the world of COTS software development (COTS is a military acronym meaning "commercial off-the-shelf"), competition is ferocious. Companies like Borland, Microsoft, Novell, Lotus, and WordPerfect are either at each other's throats or making alliances. We track each other carefully, and when one company releases a product halfway through a competitor's project lifecycle, that competitor is forced to make mid-stream changes to meet the threat, or risk shipping a product which is obsolete on the day it is completed.

As for quality, except for life-critical software, it isn't as important as functionality. Reliability is assumed, by the customer, to be in the product until proven otherwise by a specific failure. Quality *is* important, but not *as* important as time to market and getting the right feature set at the right price.

Methodologists almost always define quality as adherence to the specification. In truth, it is adherence to the customer's expectation from the time of the sale to the day he buys the next upgrade. That means we need to track customer requirements until the last possible moment before the product is released.

Specific reasons why chaos persists

We trade order for opportunity:

- *Exploiting a competitor's weakness.*
- *Leveraging a special skill on the team.*
- *Adapting an existing technology to a new use.*
- *Hitting a specific marketing window.*
- *Creating a new market.*

Things are often outside our control:

- *Engineers are variable*
- *Customers are variable*
- *Management interferes.*
- *Business conditions are poor.*

.
We aren't perfect:

- *We procrastinate.*
- *We have low expectations.*
- *We don't know any better.*
- *We fear change.*

For all these reasons and more, chaos cannot simply be laughed off in our discussions of process improvement. There will always be the need to cope with it, reduce it where we can, exploit it wherever possible, and endure it when we must.

Order and chaos in processes

Process: A pattern for solving a problem. A standard solution to a standard problem.

Processes may be mandated from outside the project, but they are generally under the control of the project team. At its heart, any project is just a series of problems which get solved. So, the concept of processes helps us talk about how best to go about that.

The concept of processes can be deceptive and dangerous. For most of the interesting problems of software development, especially in a demanding business context, or wherever humans are involved, *the processes that we talk about are usually not the ones we actually practice.*

For one thing, there is a difference between public and private process. The former is the visible element of process, i.e. the documentation and ostensible interfaces. The latter is the what might also be called the "true" process, and consists of the actual problem-solving activity, including the actual interfaces and informal communications used in the course of the work. Many private processes never get identified, and remain uncontrolled, whereas all non-trivial public processes have private counterparts which may serve or may subvert them.

As an example, management can direct a test engineer to prepare a test plan, and even provide training and a planning template. But, there's no guarantee that the plan will be a good one. Management can institute a review process, but there's no guarantee the reviewers will be any better at test planning savvy than the original engineer.

When I was at Apple, my team conducted a review of 17 test plans in use by our department. Our findings were that *none* of the plans were actually in use, at all. Some of them were almost pure boilerplate. In one case the tester wasn't aware that his product had a test plan until we literally opened a drawer in his desk and found it lying where the previous test engineer on that product had left it!

Instead of recommending that plans be better managed, we suggested that they be abolished. Obviously, if the concept of written test plans had been a useful one for us, we would have noticed much sooner that the plans weren't any good. The test plan study had revealed that test planning was being done in an entirely different way, and by different people than we had assumed. This is a classic example of how public and private processes can differ.

Problems can be solved without processes. That's called "using your head." The quality of such non-processes, as well as private processes, depends on the quality of the individual doing them, and the quality of that individual's team.

Software methodologists usually limit themselves to discussing the public side of things. They prefer the objectiveness of public processes, and they assume that private processes and non-processes can be ignored, as long as the public processes are sufficiently complete and controlled.

One textbook response to the test plan problem we had at Apple would be to redouble efforts to manage test plans, instead of looking for the private processes that were making it unnecessary for individual test engineers to use plans in the first place. Another textbook response would be to rigorously define the actual planning process— on the assumption that rigorous planning processes are cost effective in every environment.

Orderly processes maximize accountability. Chaotic processes maximize flexibility.

Private processes and non-processes are chaotic, public ones are orderly. Either kind can be applied in either a chaotic world or an orderly world.

Orderly processes: Processes which are *quantified* and *controlled* by project management, and that are performed in a *consistent* and *coordinated* fashion by the team. The dangerous assumption of orderly processes is that boilerplate solutions will efficiently solve real problems.

Chaotic processes: Processes which are *unquantified* and *uncontrolled* by project management, and that are performed in an *ad hoc* and *unilateral* fashion. The dangerous assumption of chaotic processes is that each team member will do the right thing at the right time.

Of course, there is a spectrum of order and chaos. A completely chaotic process is one which no one is aware of, perhaps not even the person using it. A completely orderly process is so ingrained into the fabric of the workplace that it, too, may not be recognized as a process. In between there are many levels of accountability and flexibility.

What ways do we have of managing these processes?

Strategies of process management

Consider three overall strategies of process management: *process control, risk management* and *leadership*. These are guiding metaphors of project management, within which everything else has a place. In fact, each of these three grand strategies includes the other ones. In operational terms, however, they are all very different.

Process control: Process control seeks to reduce variability by systematically defining and tracking every process, and taking systematic corrective action as necessary to keep the project moving toward defined goals. Process control is the epitome of order, and can only track orderly processes.

Attributes of process control	
Standardization	*Defined procedures* *Defined deliverables*
Quantification	*Quantifiable output* *Measurement systems*
Control	*Defined management process* *Corrective action process*
Coordination	*Integration with other processes*

Risk Management: Risk management seeks to optimize resources and maximize flexibility by identifying and prioritizing potential failures and deploying processes, whether controlled or uncontrolled, *to the extent necessary* to avoid the important failures. Risk management is not inherently orderly or chaotic.

Risk management drives process control.

Attributes of risk management

- *Understanding of cause and effect.*
- *Identification of risks.*
- *Deployment of necessary process.*
- *Elimination of unnecessary process.*

Leadership: Leadership deals with the problems of organization, communication, motivation and conflict resolution between people. Leadership is a very powerful discipline, and good software has been created without resorting either to process control or risk management in any systematic way. It encompasses the notion of individual commitment, initiative, creativity, and other human attributes.

Leadership drives risk management and process control.

Attributes of leadership

- *Understanding of people.*
- *Identification of goals.*
- *Deployment of people to achieve goals.*
- *Helping people improve.*

Process control is more important in an orderly world. When resources are not overburdened and the problem domain is well understood, it isn't important to continually relate every task to the specific risk it mitigates. That association can be designed into the processes when they are publicly defined (e.g. a test outline can be constructed to emphasize risk areas, then the process of following the test outline automatically takes care of risk). The issues of leadership are also not as critical, because there is less communication overhead, and less of a need for individual heroism.

Often, when chaotic processes are applied to an orderly world, the result is less efficiency, and higher risk of project failure. But, when process control is applied in a chaotic world, *the very same problem arises.*

The reason for this is the tremendous cost of maintaining consistency and coordination in a quantifiable way when the world is changing all around us. If we go to the effort of constructing an elaborate network of CASE tools, what happens when our development platform changes? What happens when we phase out C and adopt C++? Everything goes to pieces, that's what.

At a cost of $17,000, I once hired a contract programmer to create a program that would automatically check the validity of symbol tables emitted by the Borland C++ compiler. This was expected to save a lot of testing time. Two weeks after it was finished, the symbol table format had to be changed, and the tool became obsolete. Looking hard at the reason for the change, it became clear that in order to maintain the tool, I would need to dedicate someone to it about half-time, because such changes were going to happen again and again.

So much for the big savings. This tool was going to cost us a lot more money than the problem was worth.

Where process control runs into big trouble is in a chaotic world where things are changing fast. In order to maintain the controls, substantial effort must go into updating documents, distributing them, modifying measurement systems, and continuously reestablishing order.

As Figure 1 illustrates, in a world where change is a constant, it's important to use risk management and leadership to assure that no more than the essential processes are in place. As chaos increases, we must resort to the most flexible techniques we have, because highly structured methods break down.

How exactly does risk management help us deal with change?

Risk management in action
(see Fig. 3)

Risk identification...

Every defect in the product increases the risk that the product will not successfully satisfy the customer.

Risk management planning...

In a very small project, or one meant for an internal or informal customer, the risk represented by the most severe of bugs is probably quite low. So it may not be important to have a separate tester assigned to the project, or it may not be necessary to test at all.

On a particular large project, let's say we plan to perform testing, track any defects found, and assign them to development engineers.

Dialog with task planning process...

We now have a basic plan for managing that risk. To actually carry it out, we need to match our test plan to the development plan. A dialog with the development engineers is needed to coordinate this.

Risk reduction...

As the product is delivered, and we find problems, they are supposed to get fixed.

Escalation of information for analysis and course adjustment...

If they *don't* get fixed, that is a failure of the default risk reduction strategy. That triggers a dialog at the planning level once again. Either the plan is altered, or the risk represented by those bugs is called into question. In that case failure information is escalated to the strategic level.

Risk management and chaos

Risk management is a parallel effort to normal engineering tasks (see Fig. 3). It consists of three major activities:

Three levels of risk management	
Identification	*What are the risks?* *Ask "why?" and "what if?"*
Management planning	*How will we reduce risk?* *Study project dynamics*
Reduction	*Is the plan working?* *Deploy effective processes*

These correspond to the generic problem solving tasks of goal setting, task planning, and task completion. Even more generically, the levels can be thought of as strategy, tactics, and execution.

Risk management has been routinely underestimated or mistaken for "cowboy" methodology in the literature of quality assurance. Most of the reason for this is the assertion that risk is automatically managed if we simply apply the "guaranteed formula for success", mentioned above.

Think of risk management as the art of protecting the project from failure. At each level there is a dialog that goes on between risk thinking and success thinking. This often corresponds to an actual conversation between Development and QA. The dialog can be as simple as a conversation or as complex as a full blown verification process with supporting documentation, depending on the risks involved. Process control falls on both sides of the model. Through the action of the dialog between the two sides, process control is deployed as needed.

The other dimension of communication is passing down assignments (deployment) and passing up information regarding success or failure of those assignments. This escalation of status then feeds back into new plans and assignments.

This horizontal communication (between success maximizing tasks and failure prevention tasks) and vertical communication (between levels of analysis) represent a *stimulus and response* dynamic. Stimulus and response is purely situational and inherently flexible.

As you can see, risk management need not be a grand bureaucracy. It can be as simple as a way of thinking that relates goals with tasks through an understanding of project dynamics.

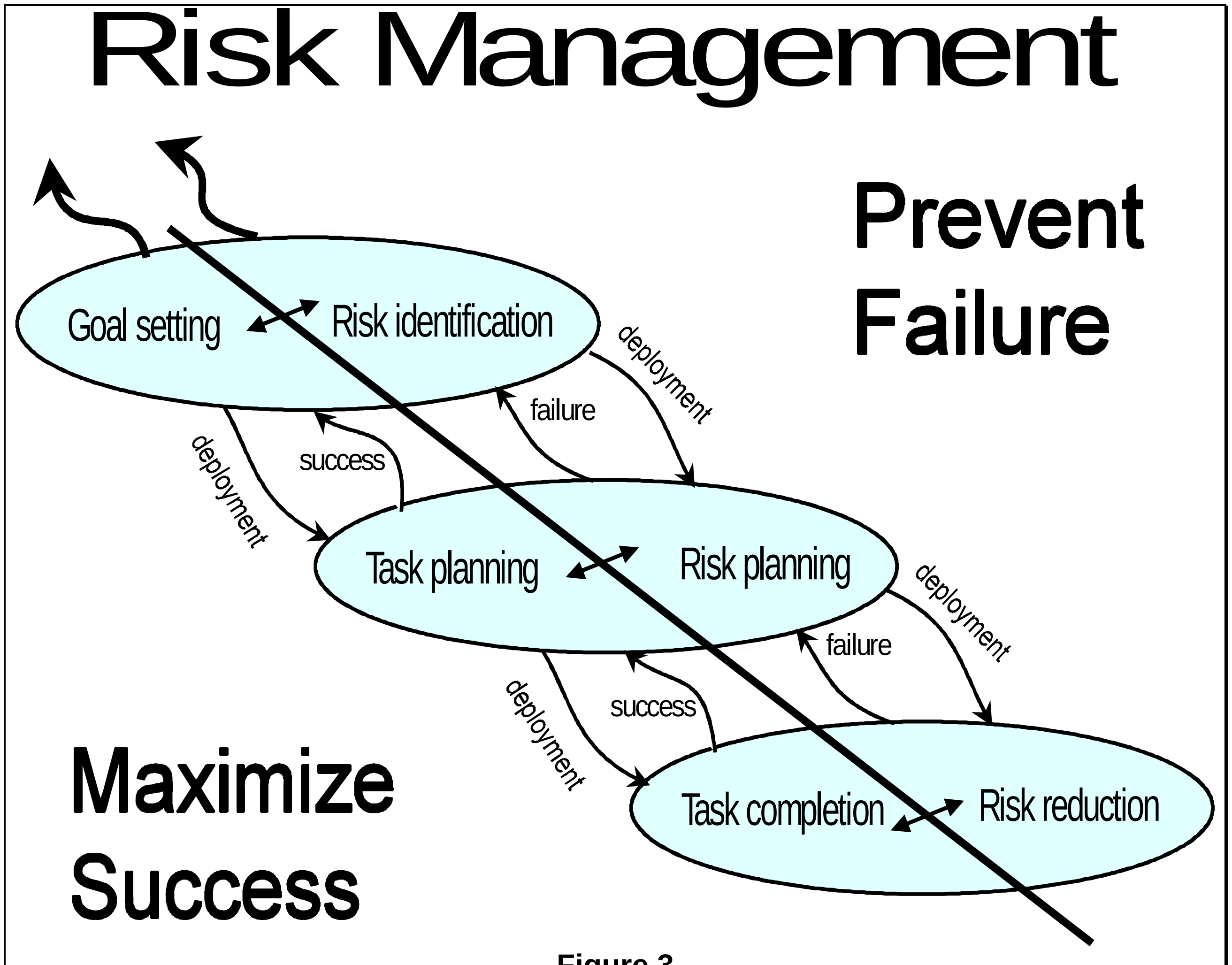

Figure 3

The risk management cycle is performed not only at the project plan level but even on small sub-tasks. The model is recursive.

Taking the case of software defects, the risk management cycle occurs at project inception, but it also occurs for each bug, as decisions are made regarding how and when to fix them. The cycle happens again as we adjust the process of bug review to meet the challenges of efficient management near the end of the project.

The hard part is spotting risks, prioritizing them, and knowing how to eliminate them. The risk management framework is easy to understand and to use, but it only sets the stage. We need good heuristics to actually perform it well and good processes to call upon when the risk demands it. How do those heuristics and processes evolve?

As an example of heuristics for risk management, I've attached the list of key questions that we ask of each defect at our bug review meetings (see Appendix). These are not the heuristics, per se, they are only references to them. They serve as aids to experienced project managers, or as a basis for training.

In the final part of this paper, we examine this list as an artifact of process and a product of evolution.

Process evolution

Contrary to popular belief, it is possible to evolve processes even without the aid of a process control authority. Even if we can't get help from management, as long as the environment isn't actively hostile, each of us can individually work toward a better organization. For a dedicated process engineer like me, it's always nice if the organization both desires improvement and commits to working on it. Still, if one or both of these elements is weak or missing, as they usually are, then a risk-based, "guerrilla" method of evolution can be employed.

The issues involved with this kind of evolution are:

Evolving a Process	
Engineering it	• *Use personal initiative* • *Focus on risk* • *Work with the right people* • *Make it tolerant* • *Experiment informally*
Deploying it	• *Build on existing processes* • *Start small. Score quickly.* • *Use mentoring, not manuals*
Maintaining it	• *Evaluate frequently* • *Formalize only if necessary*

1. Use personal initiative

Processes evolve through direct attention by a champion who maintains it until it becomes popular and widely understood. Committees aren't champions, quality manuals aren't champions. Evolution is successful when someone takes the time to experiment with a new way of doing things. If possible, appoint someone to work full-time as a coordinator of process improvement.

At Borland we invented a role called a "metrics engineer" or "QA coordinator". The mission of the metrics engineer is to maintain the best possible picture of the state of the project. That means maintaining and auditing project documentation, creating reports including information drawn from various departments, watching the development processes, and helping to evolve them.

A metrics engineer can be the focus for process experimentation and documentation, but even if nothing like that role exists in your organization, it's always possible to choose an important perennial problem and set aside a portion of time to solve it. Evolution is possible even under very difficult circumstances, even if it is often very slow.

2. Focus on risk

This is very important. If you create processes for *imagined* risks rather than clear and present ones, you won't get anywhere. Moreover, the risks must also be *accepted* as such by the team.

In the Apple test plan example, it would have been completely useless to improve those test plans in an atmosphere of complacence about planning. That whole department was phased out, a couple of years later. As it happened, the complacence was due to the unimportance of the mission that the department was given. Why go to the expense of doing great testing on unimportant software?

The way to tell if a risk is real is to look at past failures and determine how fearful the organization is that a given failure will recur. People learn best from their own painful experience. Use that as an engine for evolution.

3. Work with the right people

The tendency, when working with processes, is to determine what the right process is, and then try to paste that onto the organization. Instead, it's much more effective to work in harmony with the team. If the team wants an improved testing process, don't force a defect prevention strategy on them. Suggest it, demonstrate it, but don't push it.

At the same time, don't work with more people than necessary. Involving too many people early will bog everything down. When I have a new process in mind, I may float the idea past a few key influencers in the company, but I don't get serious about it until I have some kind of "straw man" document or tool ready to review. Even then, I prefer to find a single team that is willing to give it a try before exposing other teams to it.

Whatever the level of actual deployment of the experimental process, that is the level on which the champion of the process should involve other people. Also, bear in mind that success on a particular level may not scale up to the next level, even though local success does add momentum to the cause.

If a process is not accepted, recheck the analysis of the importance of the risks that the process is meant to address.

4. Make it tolerant

For a new process to be successful in a chaotic world, it has to tolerate being ignored for a few days or months. It must require minimal maintenance or coordination— none, if possible. It must also tolerate being performed *incorrectly*.

Everybody's busy. It's hard to assimilate new ways of working.

5. Experiment informally

Again, the temptation is to define and deploy the process rapidly. Resist the temptation. Instead, deploy an experiment. See if it works. Update it. Don't appoint committees or task forces. Don't think it to death. Don't design an exotic automated tool. Experiment with different pilot processes in real situations over a long period. That's much less threatening and cumbersome than a new "official" process that comes out of nowhere.

The most important reason to experiment rather than define is to allow room for failure. If we invest too much in immediate and complete success, we're less likely to learn from failure. Think of experimentation as "failing forward".

Another important reason to experiment is to reduce interference from people in the organization who may feel threatened by the new process. If they think they have to "speak now or forever hold their peace" they may try to kill the initiative before it's fully formed. In that situation, I've found that by treating the process as an experiment, it's much less threatening. I assure them that before any process becomes official, they will be given plenty of time to review and modify it.

6. Build on existing processes

Take stock of existing processes, structures and habits that are already in use. Connect the new process into those. There is usually a higher cost to introducing a brand new way of thinking than there is to allying with an existing point of view.

For instance, if there is a formal software build process, but no formal acceptance testing process, the natural way to introduce acceptance testing is to make it a part of the build process, such that no new software is delivered until it passes the acceptance test. This will probably be easier to achieve (all things being equal) than asking the developer or the tester to perform it before or after the build occurs.

7. Start small. Score quickly.

In my experience, most initiatives that fail seem to collapse under their own weight. That's why I stress evolution as opposed to definition or deployment. The primary sense of the word "evolve" is that of starting small and growing in steps.

There are some engineers who can only think in terms of programmatic solutions, no matter what the problem. In fact, software usually complicates things. A good way to kill a new process is to try to develop a dedicated, GUI, networked, software platform to drive it!

Instead, think of a completely manual process that can be implemented in a few days or a week at most. Use off-the-shelf software, if necessary. Only develop software after the process is proven, and even then, create and deploy it in stages.

8. Use mentoring, not manuals

Instead of formalizing processes, we should train ourselves and our teams to think situationally. That means focusing on cause and effect, and letting processes be provisional and experimental.

9. Evaluate frequently

Find ways to evaluate the effectiveness of the process. These can be purely qualitative measures, or quantitative. One very good way to do it is to relate the process to past failures to show how causes have been eliminated or reduced.

10. Formalize only if necessary

In an orderly world, formalization is great. I'm all for it. In the chaotic world, formalize only when the experimental process is fully understood and accepted by the team. Even then, plan for the expense of maintenance.

These principles, drawn partly from risk management and partly about leadership, have proven successful at bringing organizations slowly forward, even in the midst of one crisis after another.

You don't need top management support to do these. You don't need documentation or ISO-9000 certification. These are techniques like any others, requiring skill and determination to accomplish. But, unlike process control-oriented techniques of evolution (i.e. TQM, CMM), risk-based evolution can be practiced by a single person without anyone else's cooperation.

Failure is a precious resource.
It drives evolution.

You always have the option to develop in yourself the heuristics to spot risk and communicate with others about it. You always have the option to be a champion of process improvement, especially on the small scale described here.

Even when we are asked to do the impossible, some portion of that may be possible, and the experience of working toward that goal can be harnessed to improve the organization.

Example: Bug review meetings

This is an example from my experiences at Borland. Bug review meetings began in our group after a last minute bug fix introduced a major new defect which was not discovered until after shipment. The bug cost us $250,000 in remastering, disk reduplication, unpacking and repackaging expenses. This happened before I joined the company, so I don't know for sure whether it's a true story. All that matters is that a legend was born into our group of the big bug that got away.

The legend dramatically elevated the sense of risk about making changes to the product late in the cycle. Bug review meetings, which we call "bug councils" were introduced as a strategy to reduce the risk.

The idea of bug councils is for all the project leaders to look at each and every defect and suggestion on the bug list and collectively decide what to do about it. It's a change control as well as a quality standard control process.

With between fifty and one hundred new bugs coming in every day, and councils puttering at the rate of thirty bugs an hour, it's a meeting that lasts several hours each day, every day, in the final three weeks before sign-off.

How do bug reviews look against the list of evolution principles?

1. Use personal initiative

Counciling isn't a very good example of this principle, because it was a technique invented by the team leaders. The bug review questions, however, were recorded on my initiative. I wanted to help new councilors understand the thought process. In so doing, the process has been evolved a notch.

2. Focus on risk

Bug counciling is a risk reduction strategy with regard to change control. It is a risk identification strategy with regard to the quality standard, and drives risk management planning with regard to incremental test planning needed to verify fixes that have been approved.

Most of all it is fueled by the pain our team has felt when problems were introduced late in the project cycle, or when minor known problems were allowed to remain in the product that later turned out to be major problems.

3. Work with the right people

All of the leaders of the project were involved in the decision to start the formal bug reviews, and the process was shaped with their involvement. No one above the first level of management was very much involved, however, and no one from the other development organizations within Borland, although it has since propagated to several other groups.

The clear necessity of counciling, and the success we've had with it, have served to break down most of the resistance to bug counciling. The point of contention now is how early in the project to start doing it. The answer to this question may lie in better metrics to track critical bugs, or in better prevention strategies.

4. Make it tolerant

Councils are tolerant in that they are driven by the bug list. If we don't council for a couple of days, then the list grows longer, but we can't ship without performing the review (see #1), so the process is almost self-governing.

At first, we kept the current hot bug list on a big whiteboard. The board required a lot of maintenance, so a new system was adopted that was driven by the bug tracking database, requiring much less coordination. Because our bug database is a decentralized system, we were able to modify our version of it to do this without disturbing any other teams.

Another way we made it tolerant was to divide the review into stages, calling people in when their bugs were up for review. In this way, only a core group of about four leaders had to be there all the time, instead of the twelve or so which comprised the full management team.

5. Experiment informally

Bug counciling is an institution now, three years later, but they are still being tweaked. We've tried many variations on the them, including mini-councils (a leader each from QA and Development), meta-councils (full councils that meet just to assure that mini-councils are happening), feature councils (same meeting format applied to the feature set), schedule councils (reviewing anything that threatens the schedule), "T" councils (full councils that only look at bugs marked as candidates for deferral), even something dubbed the "micro-council" (The program manager reviews the list alone, seeking out QA leaders as necessary to ratify his decision).

The bug review key question list (see Appendix) is the first documentation of the lore of bug counciling, so the formalization process has begun. This specific practice has not yet spread to other divisions, however.

6. Build on existing processes.

Bug counciling is a systematic approach to something everybody has to do anyway. We all have to make decisions about how good is good and whether the risk of changing the software exceeds the risk of not changing it. We now do it in a meeting, instead of in our offices.

7. Start small. Score quickly.

Bug counciling got going very quickly after the idea was adopted. It started as an entirely manual process, with ground rules made up in the course of a few minutes.

8. Use mentoring, not manuals

There is no bug council guide, or rules of order. If ever there is one, it will probably be used for training purposes only, and not as a rule book. The question list in the Appendix is just such a tool. Although handy as a reference, it is still better suited for training.

9. Evaluate frequently

In the early days of counciling, there were frequent comments and suggestions regarding its effectiveness. With each project, we get more confident in the basic technique.

One way we evaluate it is to look at how many bugs found in the field were known to the council beforehand, and how many were caused by fixes authorized by the council. When a new kind of problem slips through, new heuristics crop up to defend against a repeat disaster. The bug review questions give an idea of how much there is to think about.

10. Formalize only if necessary

The bug council was driven to a certain amount of formalization by the sheer weight of the process, but it remains today an experimental process, subject to change and optimization.

If a substantial number of leaders leave the company all at once, the counciling concept will go with them, but failing that, there is little need to further formalize the process.

Example: Better Bug Metrics

This is a quick history of the evolution of a daily bug metrics collection and distribution system which changed the way projects are managed in my particular group within Borland.

The challenges in this area were:

- Determining what to measure.
- Determining how the metrics should be used.
- Packaging the metrics for ease of use.
- Adding a process when we already had too much to do.
- Getting the team interested in the metrics.
- Overcoming resistance in team members who feared the misuse of statistics.

It took eight months to craft the metrics system, but in small steps such that I was never away for too long from my other duties. The system was recognized at our project debriefing as having contributed to the success of the project. Now there are two large teams at Borland who are using it, one small team, and another that has requested it.

The system runs every night, queries several bug tracking databases, and produces a comprehensive text report which is then imported to a spreadsheet, where key elements are graphed. The graphs and the report are concatenated and sent via email to the managers of the project, and anyone else who asks, on a daily basis, and to almost everyone else on a weekly basis.

The system is an example of an experimental, skunkworks process, growing from the grassroots in response to specific needs. It shows that improvement can happen even without an organizational cattle-prod in the form of a documented official process, and even without a committee or blue-ribbon task force.

First, a problem is identified and a solution is suggested by a member of the team. Principles #1 and #2 apply.

Development on the system started when I witnessed disagreement among the managers in the Languages department as to how long it would take us to converge from a certain number of critical bugs to zero, so that we could ship. I thought it would be useful to graph the number of critical bugs each day after the review meeting, to see if there were any interesting trends to be seen. That might help us improve our ability to schedule project milestones.

A solution is hand generated to prove the concept. An unexpected side benefit emerges as a result of the experiment. Principles #1, #5, #7, #8, and #9 apply.

I performed the graphing by hand for 11 days in a row, each time gritting my teeth and thinking of ways to improve the process (which required a total of twelve separate queries of the bug tracking system with manual data entry to a spreadsheet, and took about 20 minutes). I sent the graph to the managers for their curiosity. Since we have to ship with zero critical bugs, the graphs were instantly popular with a couple of people who really wanted us to ship soon. They began looking for ways to drive the graph to zero. I realized that the graph could be used as a motivational instrument, both for people trying to find bugs and those trying to fix or prevent them.

The process is optimized based on experience. Automation is introduced. Principles #1, #5, #7, and #9 apply.

The milestone was reached and I stopped doing the graphs. But I knew it should be restarted for the next milestone in order to compare that data against the first run, so I asked a buddy of mine, who was familiar with Paradox database programming, to write a script to automate the data collection. While he was at it, I had him throw in a few other basic metrics just for fun.

Meanwhile, I created a new spreadsheet to provide a more comprehensive graph set and store the other data for later use.

The new process is connected to (and reinforces) existing processes. Principles #3 and #6 apply.

The program manager called me a couple of weeks later and said that we needed to increase the sense of urgency about shipping, so would I please restart the graphs. I responded that since the graphs are based on output from the bug review meetings, he would have to get those meetings restarted first, which he did.

An attempt at early formalization is resisted. The process remains identified with the originator. Principles #4 and #10 apply.

Because I wanted to entrench this new concept I came up with a name for it: "Hot Convergence Charting". The name has not caught on, yet. I'm told they're known as "James' Graphs". Outperforming the projections suggested by the graphs is called "beating James."

I also installed the graph data collector on various other manager's machines, but they almost never ran the scripts. They found it easier to ask me to do it.

A second round of optimization triggers a broader definition of the process and a higher quality goal for the system. Principles #3, #4 and #9 apply.

During the next convergence cycle, I added features to the graph in response to the way I saw it being used and the questions that came up while discussing it. While doing this, I decided to throw out the original Paradox script and create a brand new, extensible metrics gathering architecture.

I commandeered an old Dell 325 (a big heavy beast which fell on a concrete sidewalk while I was wheeling it over on an office chair, but it still booted up when I plugged it in), and made that a dedicated metrics collection machine. I put all the metrics scripts under version control.

The new architecture emerged rapidly, since I was pretty fired up at this point and trying to get ready for the third convergence cycle. By the time it began, the metrics system had reached it's final (experimental) form, and I was just tweaking it to add special reports here and there.

At that point, the development project which was the guinea pig for the metrics system was in a state of continuous convergence, so I was running the metrics system throughout the day, measuring almost everything I ever wanted to measure, plus a lot of things other people had asked for. It had evolved into a generalized defect analysis system.

An existing process is recognized as anachronistic and discontinued. Principles #2, #5, #9 apply.

The existing, clunky, weekly bug tracking statistics, which required an hour per week to produce, and were entirely paper-based, were discontinued because my metrics measured everything and more.

The process reaches a full-blown experimental form. Its simplicity and un-authoritarian nature win it support. Principles #1, #2, #3, #5, #6, #8, #9, and #10 apply.

The VP began using it to see how we were doing. He also occasionally requested special metrics.

Decisions were being made on the basis of the reports. Special effort was made to comb bad data out of the bug database in response to concerns that the graphs might be misleading. Many critical questions were posed by some managers who now found themselves under more intense scrutiny. I tried to treat everyone who approached me as my customer and almost always modified the system in response to their concerns.

Most of the managers relaxed when they saw that the metrics were not being used punitively or dogmatically. The graphs allowed us to spot certain trends and ask certain informed questions that we would never could have before. Resistance diminished, as well, because it required no effort on anybody's part to create them (except me) and since this wasn't an "official" process, but one that was entirely voluntary on the part of the people who used the reports.

The process spawns another process, which fails, and is withdrawn before it does any damage. But, the failure itself adds to the experience of the organization. Principles #2, #6, #8, #9, apply.

Meanwhile, in the midst of another disagreement over how fast we were really converging, the program manager asked me to prepare a projection based on the convergence charts. I then created a simple model to project a ship date from recent trends.

The program manager began referring to "James' magic formula" in meetings, which piqued some interest and some alarm among the managers.

On the first trial, the formula was correct within 2 days at a range of three weeks, which was good, given the number of variables that affect the actual ship date. On review, however, I determined that it was probably luck for it to be that good. On the second trial, it was way off. I decided that the formula was a failure and scrapped it.

Another team heard about the formula and I gave them a briefing, noting it's strengths and weaknesses.

The process spreads by popular demand, remaining flexible to meet the needs of it customers. The effort to automate over the past months begins to pay off. Principles #2, #3, and #4 apply.

Another large team expressed interest in the metrics, but didn't like some of the reports. So I branched the sources and created a special version just for them. Doing metrics for the second team added approximately six minutes of work to my day after the initial startup cost.

Summary:

This is an example of people cooperating naturally, solving problems naturally, and improving processes naturally, without the need for a process control authority.

No committees or quality circles were needed. No proposals or requirements documents were created or reviewed.

Should the organization decide to formalize the process, substantial experience in actually using it will be available to guide that effort.

Appendix: Key Questions For Bug Review

Problem Analysis

Frequency

1.1. How was the bug found?

1.1.1. Was it found by a user?
1.1.2. Is it a natural or contrived case?
1.1.3. Is it a typical or pathological case?
1.1.4. Was the bug caused by a recent fix to another bug?

1.2. How often is it likely to occur?

1.2.1. Is it intermittent or predictable?
1.2.2. Is it a one-time problem or ongoing?

1.3. How soon after the bug was created did we discover it?

Severity

2.1. Does the bug cause any data to be lost?

2.2. Will it cause an additional load for Technical Support?

2.3. How likely is the user to notice it when it occurs?

2.4. Is it the tip of an iceberg?

2.4.1. Will it trigger other problems?
2.4.2. Is it part of a class of bugs that should all be fixed?
2.4.3. Does it represent a basic design deficiency?

2.5. Was this bug shipped in the previous release?

2.5.1. Did Technical Support hear anything about it?
2.5.2. Has anything changed since the last version that would make it more or less of a problem?

2.6. Are there any international implications?

2.7. Does the bug break test automation?

2.8. Is this bug less severe than others we've deferred? more severe than others we've fixed?

Publicity

3.1. Are certain kinds of users more likely to be affected than others?

3.1.1. How sophisticated are those users?
3.1.2. How vocal are those users?
3.1.3. How important are those users?
3.1.4. Will it affect the review writers at any major magazines?

3.2. Are our competitors strong or weak in the same functional areas?

3.3. Is this the first release or is there an installed base?

3.4. Is the problem so esoteric that no one will notice before we can update the product?

3.5. Does it *look* like a defect to the casual observer, or more like a design limitation?

Solution Analysis

Identification

4.1. What are the workarounds?

4.1.1. Are they obvious or esoteric?

4.2. Can we "document around it" instead of fixing it?

4.3. Can Technical Support create a Tech Note to explain the solution?

4.4. Can the solution be postponed until a later milestone?

4.5. Is a fix known?

4.5.1. Are there several possible fixes or just one?
4.5.2. How many lines of code are involved?
4.5.3. Is it complex code or simple code?
4.5.4. Is it familiar code or legacy code?
4.5.5. Is the fix a tweak, rewrite, or substantial new code?
4.5.6. How long will it take to implement the fix?
4.5.7. What components are affected by the fix?
4.5.8. Will it require rebuilds of dependent components?
4.5.9. Does the fix impact doc. in any way? screenshots? help?

Verification

5.1. What new problems could the fix cause? what is the worst case?

5.2. How effectively could we test the fix, if we authorize it?

5.2.1. Was this bug found late in the project? does that indicate a weakness in the test suite?
5.2.2. Will the test automation cover this case?
5.2.3. Could the fix be sent specially to some or all of the beta testers?

5.3. How hard would it be to undo the fix, if there's trouble with it?

Perspective

6.1. How *dangerous* is it to make changes in this code?

6.2. Will a fix to this component be the *only* reason to rebuild or remaster?

6.3. How does the *overall* quality compare to previous releases?

6.4. If we think this bug is important, why not slip the schedule by *two weeks* and fix more bugs?

6.5. What would be the *right* thing to do? the *safe* thing to do?

Prevention

7.1. Was the problem caused by a previously authorized change?

7.2. What was the error that caused the defect?

7.3. Is there any internal error checking or unit test that should be added to catch bugs of this type?

7.4. Is there any review process that could catch bugs like this before they get into the build?