PART I

MANUFACTURING SOFTWARE

Software development is a twentieth century phenomenon. That century began with fifty amazing years of mechanization and increasing efficiency. Those fifty years built on something. The fifty years before saw manufacturing move from seat-of-the-pants to "scientific."

Software development has matured much the same way. When computers themselves were new, software development was like taking a stroll in an earthquake. You couldn't go half a block without the ground shifting violently. Everything changed all the time. That was a prescription for extreme management stress. As software development went mainstream, and virtually every big company had at least some sort of IT department, managers got understandably tired of that stress. They tried to bring software development under control in the same way their peers (or maybe even they, at some point in the past) controlled other kinds of production. The manufacturing mindset was born. It goes something like this:

> We build software just like we build other stuff, so the same management assumptions and principles apply.

This often unconscious, mostly unquestioned metaphor—that we manufacture software—dominates software development in most companies. It might have made sense fifty years ago. It doesn't now, but it hangs around like a bad habit. The software development industry is stuck in a rut.

Software development work is different from manufacturing work. The same old management assumptions and approaches don't apply anymore. More importantly, they don't work.

1 The Manufacturing Mindset

Projects dominated by the manufacturing mindset begin with the "figure it out" step, which has two parts. The first is a small-scale shooting war among stakeholders over what the project should be. They might argue back and forth about actual need, perceived need, budget, control, and resources. Managers do their best to figure out a way to make as many stakeholders as happy as possible, but eventually have to hope for the best and get started.

The second step is to plan. And plan. And then plan some more. This can take days or months, or years, in extreme cases. In the end, a manager has a thick plan with a budget, a schedule, and lots of details about how the project will go. That is, how it *should* go. The plan is a manager's prediction of what he thinks will happen. Most managers would agree that estimates are estimates, not promises, but somehow all the assumptions in the plan take on the air of gospel truth for other people who read them. Whether they like it or not, managers become victims of their plans.

Everybody, including the manager, knows that instinctively. They eventually feel it. As the project wears on, at some point the team probably gets a collective sinking feeling. There's too much to do and not enough time left in the schedule. Managers try to help by controlling the process more. They do their best to squelch scope creep. Jim Highsmith describes this as

management's response to "high change." Management plans to achieve some desired future state, then tries to control time, cost, quality, and feature variance to keep things on track.[1] The rub is that the project seems to get out of control anyway. Scope creeps. Teams miss deadlines, which causes other teams to miss deadlines. If the team works hard, and gets lucky more than once, they might deliver something close to what users actually want at the end. When the project is over, people probably feel a little used, and disappointed that they didn't realize the potential they thought was there when they started. Then it's on to the next project to do it all again.

Some projects don't look like this, but many do. That's why so many public and private industry studies show disheartening project failure rates. I'm not sure any two surveys agree entirely, but I think it's safe to say that, try as we might, we just can't produce the results we want often enough. We all know this story. We know before we even start projects, but we keep up the charade because of our manufacturing mindset. The result is two worlds, the one the plan talks about, and the one we live in.

Making Cars

The manufacturing mindset tells us we should

- Figure out the end product we want
- Then build it over and over again
- Then inspect for quality

When you manufacture something, the creative exercise is over by the time the assembly line starts running. You already know what you need to build and how you need to build it. All that's left is to flip the switch and go at maximum sustainable speed. Quality, although you do your best to build it in, is something you have to check the end product for.

How does an automobile company create a car? First, they spend very large sums of money to engineer it. Then they design the production process that looks something like the following (grossly oversimplified, of course):

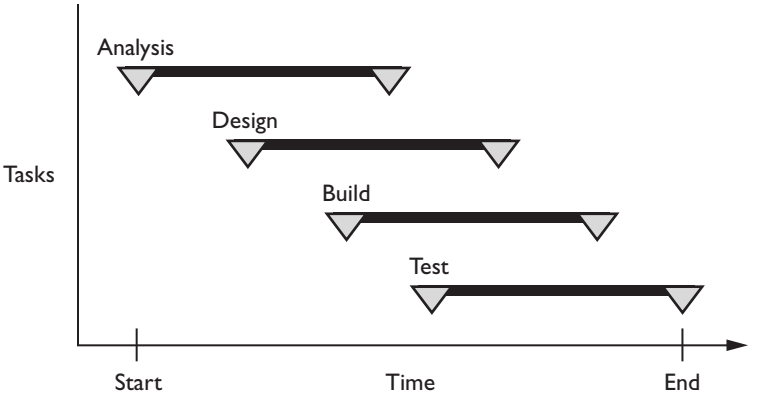
[1] Jim Highsmith, *Adaptive Software Development: A Collaborative Approach to Managing Complex Systems*, Dorset House Publishing, 2000, 183.

- Assemble the chassis
- Put in the engine
- Attach body parts (like fenders)
- Etc.

Then they run the process over and over again to churn out vehicles that are (they hope) perfect copies of one another. Then they inspect cars coming off the line to be sure they're ready to leave the factory. The whole process, from soup to nuts, looks something like this:

Making Software

Now, think about software development plans. Almost every one I've ever seen (including my own) looks like this at some level of summarization:



There's usually a large amount of detail underneath, but the rollup looks remarkably consistent. The fact that Microsoft Project® requires "phased" plans like this, and even has trouble with overlapping phases, is telling.

Notice three important things about this plan. First, there are distinct phases. It doesn't matter what you call them (phases, high-level tasks, etc.), they're distinct. Second, there's one job per phase. You might start designing things before analysis is finished, but you'll be designing the things you analyzed already. If you're designing in the Build phase, you must be doing something wrong. Third, the path through the plan is linear:

We proceed in what amounts to a straight line until we're done. We figure out the software we think we want in Analysis and Design, then we write the code for it in the aptly named Build phase, then we Test it to make sure it works, if there's time left. That makes intuitive sense. How will we know what to build if we don't figure it out first? How can we test something that doesn't exist? We treat these as rhetorical questions, make our plans, and get going. Software development in most companies bears a striking resemblance to the automobile manufacturing process I talked about a couple of pages ago.

Keep this image in mind as you read this book: We have an assembly line mentality when it comes to software development. This way of thinking, and the behavior you see on most projects, is the result of some powerful conditioning. It began three hundred years ago, and is now set in cement.

2 The Way We Think

In *Complexity and Management*, Ralph D. Stacey and his coauthors claim that three hundred years of science, philosophy, and psychology explain how today's managers tend to see their world. They call it *systems thinking*, and they summarize it this way:

> The manager…is understood as one who observes the causal structure of an organization in order to be able to control it…. This is taken to mean that the manager can choose the goals of the organization and design the systems or actions to realize those goals…. The possibility of so choosing goals and strategies relies on the predictability provided by the efficient and formative causal structure of the organization, as does the possibility of managers staying "in control" of their organization's development. According to this perspective, organizations become what they are because of the choices made by their managers.[1]

[1] Ralph D. Stacey, et al., *Complexity and Management: Fad or Radical Challenge to Systems Thinking?*, Routledge, 2000, 62.

Seems right, doesn't it? Unfortunately, this explanation is far too tidy to describe real life. There's a dirty little secret buried in there:

> Managers objectively observe their organizations at work and make the rules for everybody else to follow, which everybody else obediently does, to the letter.

Does that sound right? Let me rephrase slightly:

> Managers are objective and omniscient, and workers are cogs in the machine managers tinker with.

Does *that* sound right? Everybody, including managers, knows it doesn't. This is comic, which might explain why *Dilbert* is such a hit. It's also why systems thinking leads to simplistic explanations and prescriptions that don't make much sense in the real world. But systems thinking has become the dominant management theory in most organizations, software development organizations included. This comes very much from a single man.

Taylorism

The manager of a typical fast-food restaurant has a simple management philosophy:

> Follow the steps to produce the perfect hamburger (or burrito, depending on the restaurant).

Anyone who doesn't like regimented structure would want to off himself if he worked at McDonald's®.

For this philosophy of work, we can thank a man named Frederick W. Taylor, a foreman at the Midvale Steel Company in 1880. He noticed one overriding fact about steel production: ubiquitous inefficiency. Summarizing a bit, he suggested two broad causes:

- Managers weren't managing
- Workers were inefficient

Taylor's world was dominated by what he called "rule of thumb" management, which turned day-to-day operations over to foremen and allowed

workers to produce in whatever way they saw fit (it let them pick their jobs and use "rules of thumb" to complete them). In Taylor's mind, managers needed to take more control of the production process. Their lack of control allowed inefficiency to creep in.

Workers were inefficient for two reasons. First, they had a natural tendency to "soldier," or to lollygag but expect the same pay. Second, workers were victims of inefficient work practices that limited productivity, even under the best circumstances. Both problems were management's fault. Bad managers (or aloof ones) created an antagonistic relationship among workers and management that encouraged workers to loaf in self-defense. Those same bad managers ignored inefficiency, letting it fester until productivity disappeared.

The solution, Taylor claimed, was Scientific Management, which was based on two fundamental ideas:

- Workers should work and managers should manage.
- There is a single "best way" to do any job, and managers should find it.

Probably based at least in part on his aristocratic background, Taylor believed the vast majority of workers weren't mentally equipped to manage. They were a different breed, fit only for following instructions made by people smarter (or at least better informed) than they were. Even the smartest among them couldn't manage because they were too deep in the guts of the work to look at things objectively. Workers should have left the managing to managers, because managers were ones who had the time and ability to think about the work, rather than just doing it. And managers should take responsibility for managing effectively. That was a job most managers at the time were unfamiliar with.

Taylor believed the purpose of management was to maximize prosperity for both the employer and the workers.[2] That prosperity came from maximum productivity, meaning output. The way you got it was to discover "the one best method" for doing any particular job:

> Now, among the various methods and implements used in each element of each trade there is always one method and one implement which is quicker and better than any of the rest. And this one best

[2] Frederick Taylor, *The Principles of Scientific Management,* Dover Publications, Inc., 1998, 2.

> method and best implement can only be discovered or developed through a scientific study and analysis of all of the methods and implements in use, together with accurate, minute, motion and time study.[3]

The primary tool management had for discovering this One Best Way was a stopwatch. At Midvale, Taylor timed every activity involved with cutting metal. These "time studies" made inefficiencies obvious, which helped him know what to adjust to find the One Best Way. Once he found it for each job, he wrote step-by-step instructions on colored cards that workers would pick up in the morning. All they had to do was follow the minutely detailed instructions on the cards to go like a house afire. His 1906 paper *On the Art of Cutting Metals* recorded the results and made a science out of machining.[4]

Many people, including members of a Congressional committee that asked him to defend himself in 1912, thought Taylor's approach was dehumanizing. It was hard to argue with success, though. In their work as the Twentieth Century's first management consultants, Taylor and his followers did in fact improve efficiency in multiple businesses and industries worldwide. They changed the course of history. Populist critic Jeremy Rifkin made this grand claim:

> [Taylor] has probably had a greater effect on the private and public lives of men and women of the twentieth century than any other single individual.[5]

Manageable Problems

Taylor used a reductionist approach to problem solving, decomposing production into discrete parts that could be solved separately. Most of us find it much easier to approach problems this way, rather than trying to solve the original problem in one go. This is one of the reasons people continue

---

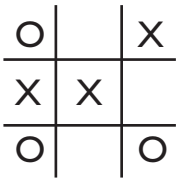
[3] Taylor, 9.

[4] Robert Kanigel, *The One Best Way: Frederick Winslow Taylor and the Enigma of Efficiency*, Penguin Books, 1997, 326.

[5] Kanigel, 8.

to apply the manufacturing mindset to modern software development, but it assumes software development is the kind of problem we like: manageable.

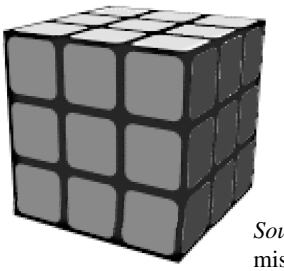
We human beings have to solve problems from the time we get up in the morning until our heads hit the pillow at night. With so much to think



Where should the next 'X' go?

about, we like our problems simple. We're comfortable solving problems like this:

These kinds of problems are pretty easy. Pay attention, give it a little thought, and the answer is obvious. Of course, this game is rarely any fun. You and I could play Tic-Tac-Toe for hours and reach a draw every time. Once you've figured it out, it's solved forever. The game hasn't changed for as long as anyone can remember.



Source: Rubik's Cube® is by permission of Seven Towns Ltd.

The Industrial Age, Taylor's time, forced us to solve more complicated problems like this:

The object of Rubik's Cube® is simple, but the solution isn't obvious: Manipulate the cube by rotating "slices" until each face has blocks of a single color. How many possible combinations do you think there are for a 3x3 cube? Whatever number you have in your head, I'll bet it's not even close. The number of possible combinations is

43,252,003,274,489,856,000

Only one represents a "solved" cube. This is definitely a harder problem, but we can still solve it. Lots of people have. You can buy a book with steps to follow. One of the first people to solve it was a fellow named Patrick Bossert—when he was 12. Many people can solve the puzzle from any scrambled starting point in less than 30 seconds (I secretly dislike these folks). Each step in the solution leads you closer to the ultimate goal, even though it may look like you're taking a couple steps backward once in a while. Once you know the finite set of possible patterns, and the steps to follow whenever you encounter one of those patterns, you can solve the problem.

Taylor's Cube

Taylor was solving his own version of Rubik's Cube™. His problem had a few primary characteristics.

First, Taylor or some of his friends could both formulate and solve the problem. They made a business out of it. It was possible to find an equation accounting for the variables that dictated how much work a man could do in a day at a particular job. At Bethlehem Steel Company in 1900, Taylor had one of his protégés, a mathematician named Carl Barth, try to devise "the law" of pig iron labor. Barth puzzled over it for some time, but eventually came up with 43% work, 57% rest.[6] Nice and scientific.

The equations describing the "laws" of work were certainly complicated algebra. They took some figuring, and they were tough to do by hand. Still, they were solvable on special slide rules pioneered by Barth.

Second, Taylor's problem was static. In various ways and in various places, Taylor worked on steel production and related problems for almost thirty years. There were a few dominant players in the industry, and the competitive environment changed slowly. Most companies who made and tooled steel did it substantially the same way, had substantially the same limitations, and could benefit from the same "laws" Taylor wanted to install. This stability meant Taylor's problem was really an optimization exercise of finding the One Best Way. Stacey and his colleagues described scientific management like this:

---

[7] Taylor, 27.

> There is a quite explicit assumption that there is some set of rules that are optimal; that is, that produce the most efficient global outcome of the actions of the parts, or members, of the organization.[7]

Third, Taylor's problem was predictable. Once you discovered the One Best Way for each job in the factory, you could simply plug in values for the variables and predict your optimal productivity level and output. Plug in the numbers and get the predictable results. Sometimes the results were counterintuitive, but the results were still predictable, once you knew the equations. This way of thinking has permeated Western society, and IT organizations are no exception.

Taylor's problem required a slide rule. It was complicated, but it was still one you could get your head around if you kept at it long enough. And that's the kind of problem software development managers can manage in the way they've been taught. As soon as the problem deviates from that, traditional management starts to fail. Life gets very uncomfortable very fast. So most managers do the natural thing and avoid that pain, even though it's a bad choice.

Doing What "Works"

Why do drunks keep on drinking? In *Life Strategies*, Dr. Phil McGraw explains that we continue to engage in self-destructive behavior because it "works" for us, meaning it gives us some sort of payoff. At some conscious level, we may know that our behavior isn't really working, or causes pain, but we must be getting some payoff or we "wouldn't do what [we] do or accept what [we] accept."[8]

Drunks stay drunks because it works for them. Their desire to drink can make them oblivious to the pain it causes other people, or let them breeze right by it on the way to the bar. Their "personal problem" is their own fault, and they're responsible for it, but they're not alone. Other people in a drunk's life are letting him get away with it. They're allowing his drunkenness to work for him by providing a safety net to keep him from getting embarrassed or hurt. As long as that safety net is there, he probably won't

[7] Stacey, et al., 62.

[8] Phil McGraw, *Life Strategies*, Hyperion, 1999, 93.

quit. Until a drunk realizes that he is one, wants to quit, and has to quit, but can't, he won't.

Why do we overeat? Why do we treat other people badly? Why do our organizations eventually squeeze out the capacity to change? Why are our software development plans unrealistic? Why is a good bit of our software short-lived junk? Because it works for us. Tom DeMarco says,

> When you find yourself applying the same response no matter what the stimulus, you're no longer acting in an entirely rational way. Your behavior is more characteristic of addiction.[9]

Treating software development like manufacturing—assuming it's an efficiency optimization problem—is a symptom of our addiction. That addiction, self-destructive though it is, continues to work for managers. Their reward for the manufacturing mindset is reduced anxiety.

Being Pavlovian

Ivan Pavlov won the 1904 Nobel Prize in Medicine for making dogs drool. He did a famous set of experiments in which he rang a bell before feeding his dogs. Over time, the dogs were conditioned to start salivating whenever they heard the bell. He could just as easily break the conditioning by ringing the bell without giving the dogs any food. This experiment and others by Pavlov paved the way for the behaviorist psychology pioneered by John Watson beginning in roughly 1913.

The premise of behaviorist psychology is simple: People do what they get rewarded for. Unless we're getting a payoff, we'll stop. Our payoff for making patently absurd software development plans is decreased anxiety and increased safety. I give managers a lot of credit. Software, in many ways, is inherently unmanageable. Trying to manage it anyway is a challenge. Managing it well is even harder. Most of the time, managers understandably fall back on behaviors they get rewarded for. Conditioning is powerful.

In an overwhelmingly chaotic environment, software development managers crave predictability. The only way to get it is to bring more stuff

[9] Tom DeMarco, *Slack,* Broadway Books, 2001, xiii.

under control. In *The Paradox of Control in Organizations*, Philip Streatfield says feeling in control reduces the anxiety we feel when we don't know the future:

> …we project our needs for control onto our organization in the hope that it will provide a sense of being "in control" and so defend us against anxiety.[10]

Trying to solve a problem that defies control is unsettling. Problems like that are slippery, hard to think about, and even harder to solve. Managers have enough to worry about without having to stress out at work, so they tend to do what comes naturally. They try to control more, hoping that this will make life more predictable. If they don't, or can't, do that, they look for other people to do it. Somebody has to be in control, or managers will go nuts or freeze up with fear. Edward de Bono said in *Simplicity*,

> With eleven pieces of clothing to put on when you get up in the morning there are 39,916,800 possible ways of getting dressed. There are eleven choices for the first piece, ten for the next—and so on. Life would be very slow and complex if you had to figure it out every morning. So routines simplify life both as regards perception and also as regards action.[11]

Chaos is unsettling, so managers fight it the only way they know how. They take the bull by the horns and try to establish patterns, routines, predictability. When managers in software organizations try to do this, the result is the typical corporate software project. This makes them feel less anxious about the future, as they convince themselves they control it. This is exactly what the manufacturing mindset tells them to do, and they're in good company when they do it. It's the only alternative they see, and staving off anxiety by increasing the amount they control works to a point. Eventually, though, most human beings know they can't control everything. Software development managers can't eliminate all their anxiety, so they fall back on making themselves safer.

---

[10] Philip J. Streatfield, *The Paradox of Control In Organizations*, Routledge, 2001, 8.

[11] Edward de Bono, *Simplicity*, Penguin Books, 1998, 21.

Plausible Deniability

Plausible deniability is a code phrase politicians use to mean being able to claim ignorance believably when something goes wrong. That makes a politician safer, since he can't be blamed. A harried, mortal, non-psychic software development manager's plan is his plausible deniability.

When a project fails, somebody who put up money for the thing usually goes on the warpath. Heads might roll. Managers know they can't know everything, despite their valiant efforts, so they understandably want to protect themselves from the negative results of potential failure. Having a plan that makes it look like they did in fact think of everything makes explaining failure easier. There are as many explanations as there are people to offer them, but I'd guess most explanations amount to blaming

- Incomplete knowledge, or
- Inadequate control

If somebody higher up the chain asks a manager to defend himself after failing, that manager can always fall back on those two things. He can say, "Look at my plan! Look at those control mechanisms! Can't blame a guy for trying." Then he can promise to batten down the hatches, try to know everything and control everything, and have another go.

In the end, most human beings would rather be safe than at risk and only potentially successful. Many software development managers are the same, and I can't blame them. They believe control means knowing everything and having everything go according to plan. This makes them more comfortable. If they make a perfect plan, doing the work is simply a matter of following it, which is much simpler and safer. Nothing reduces anxiety like feeling safe.

But this is a false sense of security. All of a manager's planning that he thinks will reduce anxiety is based on the idea that software is like manufacturing. He's going with the flow, doing what everybody's doing because they've been taught that it makes sense. Only it doesn't.

Software development in most modern contexts rarely looks like manufacturing at all. It's different work. Once managers realize that, the foundation of the manufacturing mindset starts to shake. The only way they can realize it is to be open to the idea that there might be a better alternative. Managers need to see that the manufacturing mindset doesn't really work for them quite as well as they might think it does.

3 Different Work

Taylorism, and the manufacturing mindset based on it, makes sense for efficiency optimization problems like manufacturing. It makes very little sense for other kinds of problems. As it turns out, software development isn't like manufacturing at all. It's a different kind of work to solve a different kind of problem, despite what their learning and training tells most managers. The manufacturing mindset simply doesn't apply to software development. Software development doesn't fit.

Bad Logic

Taylor wrote *The Principles of Scientific Management* in 1911. By the 1920s, his theories and practices had revolutionized industry. By the 1950s, about the time software development made its debut, no one really argued much about Taylor's theories. People simply accepted them as fact, and as the standard for management. They added some bits to create a more humanizing workplace, but Taylor's underlying principles remained. In his biography of Taylor, *The One Best Way*, Robert Kanigel describes Taylor's influence this way:

> "[S]cientific management," as well as its near synonym "Taylorism," have been absorbed into the living tissue of American life….Taylor's thinking, then, so permeates the soil of modern life we no longer realize it's there.[1]

Whether anybody ever made a conscious decision to apply scientific management to software development is tough to know, although it could have happened. In any case, efficiency optimization was ingrained in the American psyche by the time software development came along. People took it for granted that the same thinking and management practices applied, and would produce the same results. They began managing software development like machine tooling, and software developers like steel workers.

If anybody had forced managers to say why they assumed the manufacturing model would work, I imagine their logic would have been:

- We produce stuff.
- We produce software.
- Efficiency optimization works well for other stuff we produce.
- Therefore, it should work for software development.

In a scene from Monty Python's *The Holy Grail*, King Arthur happens upon a gathering of townsfolk who want to burn a witch. Sir Bedevere, apparently the law in that town, wants to be sure the citizens are positive she's a witch. He gives them some logic to follow:

- You burn witches.
- You also burn wood.
- Witches burn because they're made of wood.
- Wood floats.
- Ducks float.
- If the accused lady weighs the same as a duck, she must be made of wood, and is therefore a witch.

Software managers over the years weren't that silly, but their logic was just as bad. The manufacturing mindset dominates because, bad logic or no,

---

[1] Robert Kanigel, *The One Best Way: Frederick Winslow Taylor and the Enigma of Efficiency*, Penguin Books, 1997, 6-7.

it's natural for managers to think that way. They don't question it because it's just the way things are…and always have been. It was natural to begin with, and constraints on managers reinforced it. Engineers told them the cost of change was high, and software development managers certainly saw that often enough. Given a predisposition to the manufacturing mindset, the idea that predicting things better and controlling them more effectively would reduce cost made good sense. Thus, management attempts to make McSoftware.

The manufacturing mindset is consistent with the way most software development managers learn to manage projects. It's natural. It has become a habit, and people tend not to question habits. But it doesn't make much sense for solving the real problem, because it's based on some assumptions that don't apply.

Bad Assumptions

The manufacturing mindset assumes that

- The problem is predictable
- The problem is controllable
- The focus should be on optimization

When those things are true, it makes sense for software development managers and their teams (as the manufacturing mindset tells them) to understand the overall problem, to design the best solution, and to implement the design efficiently. Then they should repeat the process to make it maximally efficient. In this way of thinking, optimization is king, and we've carried it straight over to software development. Software development managers have been taught and encouraged to treat software development like an efficiency optimization problem. They got suckered.

I don't hear too many people claiming to use a waterfall process anymore, probably because they fear being laughed at, but consider how people in most software development organizations do things. They still make software by saying they'll understand the problem by creating (they hope) an exhaustive functional specification, by designing the best possible system to meet the spec, by implementing the design through maximally efficient coding, and by repeating and refining the process. This is the manufacturing mindset applied to software, and you can see the same three assumptions:

- That we can make useful, detailed, accurate long-term predictions about software development.
- That we can control events and people to make our predictions come true.
- That our attempts to control will produce the results we want.

The first assumption is critical. If it's not true, the others are moot (more on them in the next chapter). The only way the four-step process of the manufacturing mindset makes any sense is if we can predict the future very well. When you look closely at the typical software development plan, you can see that assumption in there.

Planning for Predictability

The manufacturing mindset tells managers they can predict the software they want and then build it.

Why would they assume that? Consider real life. What can you predict there? Very little. The norm is *un*predictability, for everything from stock prices to the weather, but most managers try to predict the smallest details of their software projects anyway. You can see this if you look carefully at the typical software development plan again. It assumes we can predict the future far enough in advance, with enough detail, and with enough accuracy to plan for almost everything.

Most plans contain sections dealing with scope, contingencies, and assumptions. The names might be different, but all that stuff is usually in there. Reflect on what those sections mean. Before we've done any work at all, we try to write down our theories about how it will go, what we'll end up with, and what might bite us in the rear end. That, of course, assumes we can know what software we want after we think about it for a while. We can then proceed to build it.

How does the first assumption of the manufacturing mindset as applied to software—that managers can make useful, detailed, accurate, long-term predictions about software development—fare? Not very well. What software will users end up wanting in six months, or a year, or two years? Ask me then. Until then, I can't say with absolute certainty. No manager or software developer can either. But the plans most managers make demand it. A manager can't make entries in project planning software unless

he knows lots of specific details about what will happen, and when, and how long each step will last.

Regardless of what we say about software projects, our plans reflect what we truly believe, or at least what plan *readers* believe. If that weren't so, somebody somewhere would say, "Hey, this is wallpaper!" Then people would stop making plans like this. Given that folks still do make them, we must really believe that software development is like factory work when, in fact, it just isn't.

A Different Kind of Problem

The manufacturing mindset applies when a problem is predictable and controllable, and when optimization is king. In other words, it makes sense when Taylorism applies. His problem was formulatable, static, and predictable, once you knew the One Best Way. Of course, since the problem was static, you could *find* the One Best Way. That made the manufacturing mindset a perfect way to think about the problem he was trying to solve. Software is entirely different.

First, software isn't formulatable. Certainly both Taylor's problem and software are complicated. The difference is, the software problem doesn't always have the same form. It rarely follows exactly the same rules twice. The act of solving the problem, or creating the software, changes the problem you're trying to solve. You can't develop a rigidly defined equation that will give you the optimal software each time.

Second, software development is a dynamic reality. It changes as it unfolds because the environment is always different. No two projects are the same. Often, the people are different, the business environment is different, and the constraints are different. Perhaps even more frustrating, the target is moving. The software you thought you wanted at the beginning of the project is rarely the software you want at the end, or even half-way through. The act of creating software influences the software you create.

Third, software is *un*predictable. The same piece of software would never emerge from the same project the same way twice, if people were willing to try that experiment. Alan Turing's application of Godel's *undecidability theorem* to computation tells us there will always be algorithms that do things we can't predict in advance. As Mitchell Waldrop said in *Complexity*,

> …any piece of code that's complex enough to be interesting will always surprise its programmers.[2]

In a word, software development is messy.

Messy Problems

When we create software, we're dealing with a problem that is

- Complicated, but not able to be formulated or solved in the way we're used to
- Dynamic
- Unpredictable

We loathe these kinds of problems. They're ugly, and they keep us up at night.[3] When we see a problem like this, we do what comes naturally, and what we've been taught to do. We simplify it. Doing that can make things worse.

We cause ourselves trouble when we try to declare a messy problem to be a simple one, just because we like that kind better. As human beings, we're all subject to the same desire to find rationality, predictability, regularity. This makes us feel more comfortable. University of Maryland researcher James Yorke believed many scientists in many disciplines had learned to ignore the messy reality around them. He said,

> They're running a physical experiment, and the experiment behaves in an erratic manner. They try to fix it or they give up. They explain

---

[2] Mitchell Waldrop, *Complexity: The Emerging Science at the Edge of Order and Chaos*, Simon and Schuster, 1992, 282.

[3] Horst Rittel and Melvin Webber described problems like this in a public policy context. In their 1973 paper, "Dilemmas In a General Theory of Planning" (*Policy Sciences*, Vol. 4, 155-169, Elsevier Scientific Publishing Company, Inc., Amsterdam, 1973), they called these problems *wicked*, and claimed the "classical systems approach" to solving them doesn't work. As they put it, "one cannot first understand, then solve." The act of solving a wicked problem affects the problem you're trying to solve. In many cases, the solution you come up with ends up being, "OK, let's try that."

> the erratic behavior by saying there's noise, or just that the experiment is bad.[4]

People in all walks of life make the same mistake. Why do investors assume cycles in gold and silver prices? Yorke suggested it's because that is the most complicated *orderly* behavior they can imagine. This is the kind of problem we see in textbooks. Problems like that are solvable. They behave.[5]

When we're faced with extremely complex problems, we often try to make simplifying assumptions, to reduce them to problems that we're more comfortable with. The story goes that a chemist, a physicist, and an economist were stranded on a remote island. Food was running low and they were getting worried. One day, to their great joy, a crate of canned vegetables washed up on the beach. Their problem was stark in its simplicity: How do we open the cans? The chemist attacked the problem with the rigors of his discipline, but got nowhere. The physicist gave it a shot, thinking he was smarter than the chemist, but he came up empty too. Then the economist almost gleefully said, "I've got it! Assume a can opener…."

Simplifying assumptions can make things easier to think about for a while. When you start believing the simplified version is the real problem, you get stuck with perpetually closed cans. The manufacturing mindset is the grand simplifying assumption of modern software development. It tries to make something messy seem neat and tidy.

Software Development

Our manufacturing mindset is great for efficiency optimization problems. For those kinds of problems, you can assume you know the answer before you start working. The actual work is just a matter of implementing the solution you've already figured out. In that case, it makes perfect sense to implement the solution as efficiently as possible. Is software like this? Not even close. Software is not solvable in the same way. What we need changes as we create what we thought we needed. What we end up with isn't what we expected. This is good, because most of the time we don't know exactly what we want when we start.

---

[4] James Gleick, *Chaos: Making a New Science*, Penguin Books, 1987, 68.

[5] Ibid.

As humans, we have a tendency to interpret everything we see in terms of things we understand. This leads us to the wrong solution over and over again. We've done the same with software for too long. Creating new software is not about efficiency optimization. We're trying to solve the wrong kind of problem when we assume it is. Software development is knowledge work. It isn't the kind of work where managers can figure everything out in advance, and then just give other people instructions to follow. It's an activity where *people* are trying to solve the problem, and that requires everybody involved, not just managers, to think. As Tom DeMarco said in *Slack*,

> Knowledge work is less like the jobs that Taylor was studying, and more like the job that he himself was doing when he studied them. It involves invention, abstraction, articulation, and skillful management of many human relationships.[6]

The manufacturing mindset started with Taylor but is alive and well today in management fads that don't deliver very often or very well (Business Process Re-engineering, Capability Maturity Model, etc.). Real life is different from what we have imagined it to be. The manufacturing mindset probably made sense when software development looked very similar to factory production. It simply doesn't apply anymore. Software development isn't like manufacturing at all. It's different work, and the manufacturing mindset was never intended for it.

But most managers see no alternative, because nobody gives them one. They're stuck. They think they have to choose between utter chaos on their projects, and forcing software development to fit within the manufacturing mindset. Chaos seems unwise, and more than a little irresponsible, so they opt for the forced fit. This is understandable, but it's also delusional, because it doesn't work.

---

[6] Tom DeMarco, *Slack,* Broadway Books, 2001, 106.

4 Square Pegs in Round Holes

Software development doesn't fit into the manufacturing mindset. Since most managers see no alternative, they're forced to try to make it fit. This is pounding a square peg in a round hole. It might go in after some serious abuse, but it'll be one splintered peg.

The manufacturing mindset assumes managers can make detailed, accurate, long-term predictions. It then assumes managers can control things and people such that those predictions come true. Managers do that because they assume control will get them the results they want. Well, the Emperor's buck naked.

Swingin' Shillelaghs

Napoleon Bonaparte supposedly said that history is a fable agreed upon. Typical software development project plans are the same, only they're more malignant than that. They're fables everybody in charge seems to mistake for truth. You'd think the Gantt chart was the Ten Commandments, with the same consequences if you disobey.

I'll never forget making my first work plan. I looked at what my team had to do, saw how many people we had, guessed about wind direction, and came up with an approach I thought made sense. It called for everybody on the team, including me, to work ten hours a day for the next two months. The date and scope were non-negotiable, so long days were the only thing I thought could bring us home on time. It seemed like a simple math problem.

When I presented the plan, the managers on the project said something like, "You can't put more than eight-hour days in the plan! Ha, hee, howl! You dunderhead!" Delivery date and scope were fixed, but they told me to cut the hours. My team delivered late, of course. During the project, though, we were flogged with the unrealistic plan, because we were deviating from it by falling behind.

Many managers use their plans to apply the pressure they think is necessary to get their teams to work harder. I can empathize, especially when the project's staring a critical deadline in the face. But in the hands of some managers, the project plan becomes a tool for severe beatings, especially under pressure. The plan usually means the schedule, and the schedule is usually "aggressive." Most software professionals know to run for cover when they hear that word. Each person on the project needs to track his time against tasks so that the project manager(s) can see deviations and squash them. As Tom DeMarco says in *Slack*, these aggressive schedules are patently absurd, but expected.[1] Companies with overcommitment built into their culture drive managers to create these things, and then pummel their people with them. Inevitably, it's the poor folks doing the work, not the managers, who shoulder the blame for not meeting the schedule. The idea that the *schedule* is bad isn't considered, but it should be:

> A bad schedule is one that sets a date that is subsequently missed. That's it. That's the beginning and the end of how a schedule should be judged. If the date is missed, the schedule is wrong. It doesn't matter why that date was missed. The purpose of the schedule was planning, not goal-setting. Work that is not performed according to plan invalidates the plan.[2]

---

[1] Tom DeMarco, *Slack,* Broadway Books, 2001, 54.

[2] DeMarco, 57.

Managers work hard to make sure reality conforms to the plan. One would hope all this sweat would pay off in better results. Most of the time, it's wasted effort, but most managers do it anyway because it's the only tool they have.

Increasing Predictability

The manufacturing mindset gives managers only one tool to make their world better: *control*. The purpose of control is to make things more predictable. Predictable things are controllable. Unpredictable things aren't, because they're too slippery. The manufacturing mindset's solution? Make things more predictable so control will work. The typical software development plan is designed to make the future more predictable, then allow a manager to control things to that future. But control makes sense only if you're solving a problem exactly like one you've solved before, in exactly the same way, to produce exactly the same results. This is a static problem, the kind the manufacturing mindset assumes.

Imagine an automobile assembly line. There isn't any room for spontaneous creativity there, and that's good. Workers should assemble the chassis, attach the body, install the fenders, and so on, in that order. Variation would cause the process to grind to a halt and reduce output. Each digression means you're headed away from your goal.

Software development is rarely static. Maybe it is for the tenth installation of a software product that requires minimal customization, but most software development isn't like that. New software development almost never is. Software applications aren't like cars, and software development isn't like an assembly line. If a problem is predictable, you can control it to make it more predictable. You can optimize something like that. But if the problem isn't very predictable, control makes little sense. Imagine trying to control an earthquake. No way. Software isn't an earthquake, but it's almost that unpredictable. Control won't work here. Not that that stops most managers, including me, from trying to control things anyway.

Despite their wishes to the contrary, managers can't control software development to the degree they want. Attempting to control things so they stay in line with managers' predictions can have unforeseen consequences that make things get more out of control. What's worse, even if nothing unexpected ever happened in software development, trying to control it would still be nuts for one simple reason: It doesn't work. The sweat managers put into controlling things doesn't produce the results they want, which only tends to get them in more trouble.

When Control Doesn't Work

Control works when it produces the result you want. It doesn't work when it doesn't produce the result you want. Not profound perhaps, but frequently ignored.

In a manufacturing environment

- You know exactly what result you want before you start the process
- Controlling things gets you the result you want predictably

What does Chevrolet Suburban number 10,147 look like? If it's worth all the hype, Six Sigma says that vehicle had better look so much like the first 10,146 of them that you really can't tell them apart. Control can work in a manufacturing setting because it can lead you to the result you know you wanted when you started.

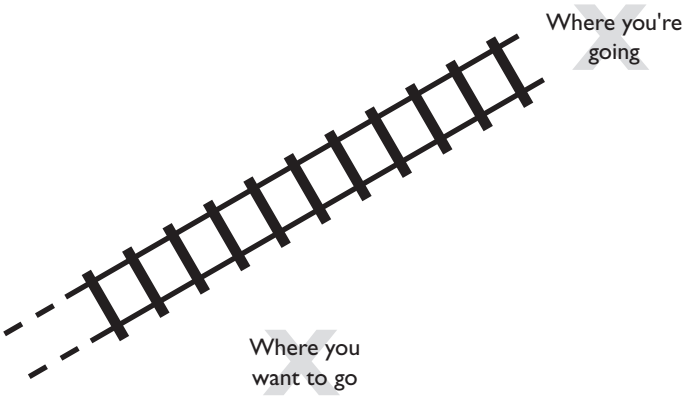
In software development, the development team doesn't know exactly what result they want before they start. They might know what they think they want, given their current limited information, but quite often they can't know what they'll end up wanting. Shoot, at the beginning of a project the people who'll be using the software don't know exactly what features they want, or what those features should look like. If they don't know, a manager and his development team certainly can't know. In my career as a software developer and consultant, I've never worked on two systems that were mostly the same, much less identical. When managers try to control new software development like the manufacturing mindset says they should, their control efforts don't have a base to stand on. But they usually aren't given an alternative, and this way of thinking is the devil most managers know, so they try anyway.

And what about the results? In a manufacturing environment, when managers tweak this or that manufacturing cell, or buy faster machines, or make the tolerances finer, all in the name of control, they move closer to optimal output. This makes sense when the target doesn't change, and when control efforts have predictable effects. Translating the same thinking to software development is dangerous.

When managers under pressure try to control software development, the consequences can be unpredictable. As an extreme example, consider a manager who tries to squeeze just a few more hours out of his people every week to get the schedule back on track. About halfway through the project, everybody on the team decides there's no more blood to give, and they all quit. Now the manager's schedule is blown to bits. A small attempt

to control more had a large, and disastrous, effect that the manager probably couldn't foresee clearly. Software development defies control in the sense demanded by the manufacturing mindset.

The manufacturing mindset tries to put the software development process on rails, like a train:



What happens when people on a team realize halfway through the trip that where they really want to go is about a hundred miles from the track? They can't get there. They have two choices: Go where they started going, or stop, which is what happens when a project gets cancelled. The manufacturing mindset often encourages managers to control their teams right to a destination nobody wants. Industry certification processes encourage exactly this behavior.

Certified to Control

The International Organization for Standardization (ISO) publishes quality guidelines for organizations. The ISO-9000 family of standards

> …represents an international consensus on good management practices with the aim of ensuring that the organization can time and time again deliver the product or services that meet the client's quality requirements. [3]

[3] All of the quoted material in this chapter concerning ISO standards comes from the ISO website at http://www.iso.org.

So it's safe to say that the ISO standards represent a significant body of management thought about what good management means.

ISO recognizes that "quality" can mean different things to different people. In the interest of clarity, they define what "quality" means for ISO standards:

> In plain language, the standardized definition of "quality" in ISO 9000 refers to all those features of a product (or service) which are required by the customer. "Quality management" means what the organization does to ensure that its products conform to the customer's requirements.

The list of Quality Management Principles that underlie the ISO-9000 family of standards are valid principles for delivering what customers want. Things like continual improvement, decisions based on facts, and a constant customer focus. It would be difficult to argue against those. You might call those principles motherhood and apple pie. The content of standards, or the principles they're based on, isn't usually what sticks in my craw. My gripe with standards like this is that they perpetuate the manufacturing mindset, cloaked in respectability.

When ISO applies the principles to software development (with standard ISO-9003), you don't have to look too closely to see that the entire standard is based on control. The word "control" appears over 120 times. Here are a few examples:

> Control your software development project and make sure that it is executed in a disciplined manner. [Subsection 4.4.1, Software Development]

> Prepare a software development plan. Your plan should be documented and approved before it is implemented…. Your software development plan should…identify important project assumptions…. present your project schedule…. define project phases and dependencies…. specify project time lines and milestones…. define each task…. describe the inputs for each task…. [and] describe the outputs for each task. [Subsection 4.4.2, Software Design and Development Planning]

> Develop procedures to control design changes that may occur during the product life cycle. Your procedure should ensure that you…document the design change…evaluate the design change…justify the design change…verify the design change…approve the design

> change…implement the design change…monitor the design change. [Subsection 4.4.9, Software Design Changes]

ISO claims its standards can apply to any business, regardless of size or industry, because "ISO 9000 lays down what requirements your quality systems must meet, but does not dictate how they should be met in your organization…." That's a half-truth. A standard makes an implicit assumption when it tells a manager that the only valid plan (if he doesn't want to cut corners on "quality") is one that he develops all at once before the project starts: First you do all your planning, then you follow the plan. That's the manufacturing mindset in action. Only now, the manager can say, "Look! We're ISO-9003 certified!"

Management standards can be an irresistible temptation for stressed-out managers to become control freaks. But the standards bodies don't recognize that. Standards bodies merrily go along their way, not questioning the last hundred years of management conditioning. The result is more pain for the managers they're trying to help. If the ISO standards and other similar certifications (like the CMM levels) are based on control, the harried manager is left to assume, as most do, that control is possible. That's true only some of the time.

The Software Uncertainty Principle

Werner von Heisenberg, one of the first quantum mechanics, developed his Uncertainty Principle to express the impossibility of describing both the position and the momentum of an electron in its orbit simultaneously. He said, "The more precisely the position is determined, the less precisely the momentum is known in this instant, and vice versa." In layman's terms, Heisenberg's principle means that the act of observing something changes that thing in unpredictable ways. This is because the only way I can know where an electron is, is to bump into it. That disturbs the system and fixes things so I can't know the exact position and momentum of a particle for sure at the same time.

Being certain about the future makes management easier. Most managers would love certainty, if they could get it. They usually can't. So they try to be reasonably sure, and have a backup plan. But this desire for control produces an ironic outcome. Without even knowing it, managers often control the process right into the ditch. I call this the Software Development Uncertainty Principle:

> Attempting to control software development interferes with the natural process that is going on, and throws it off track.

The outcome can be fatal for projects.

Two facts about software development make trying to control it a questionable management tactic:

- We can't predict all the results of our control efforts.
- We don't know exactly what result we'll end up wanting.

Software development is exploration. Control doesn't work when we're trying to solve an unexplored problem. People trying to solve the problem might have some experience they can bring to bear on it, but there isn't a map to the buried treasure. Folks on the team have to depend on their pathfinding skills.

Managers have been conditioned to ignore this. They go ahead and make their plans, thinking that their attempts to control projects are helping. As the Software Uncertainty Principle suggests, their interference hurts more than it helps.

We can't make detailed, accurate, long-term predictions about the future. We also can't control like we think we can. Even when we try, we typically succeed only in controlling ourselves more or less directly to a result we end up not wanting. That puts software development managers in a tough spot. All of their conditioning, and most of their incentives, reinforce the manufacturing mindset. It has become the accepted wisdom that people don't question anymore. Until we challenge that wisdom, we won't be able to change.