Lean Development
and the
Predictability Paradox

Mary Poppendieck

Lean Development & the Predictability Paradox

Why is it that software development projects seems to have so much difficulty delivering predictable outcomes? It seems that all too often projects are late or over budget or they deliver the wrong system or all of the above. How can the predictability of software development outcomes be increased? This executive report discusses a dilemma: in our zeal to improve the reliability of software development, we have institutionalized practices that decrease, rather than increase, the predictability of outcomes.

If this claim leaves you skeptical, consider that a decade ago, Japanese automakers routinely developed new model cars in $1/3^{rd}$ less time, for half the cost of Detroit automakers. Only one sixth of the Japanese development programs were late, compared one-half of the US programs. Yet the automobiles developed through these faster, cheaper programs were very successful. For example, Toyota's profitability from 1982 to 1998 was $1/3^{rd}$ higher than Chrysler's, twice that of Ford, and three times that of General Motors.

Japanese automakers use concurrent engineering to develop products. Light on process and heavy on communication, concurrent engineering has been widely adopted as the preferred vehicle development approach in recent years. The paradox of concurrent engineering is that development starts as soon as a vehicle concept is approved. It proceeds at an aggressive pace, even while multiple options are explored. And yet, commitment is delayed as long as possible. The result? Better quality, lower cost, faster response to the market, and more predictable outcomes. A good deal all around.

The Predictability Paradox

The best way to achieve predictable software development outcomes is to start early, learn constantly, commit late, and deliver fast. This may seem to cut against the grain of conventional project management practice, which is supposed to give more managed, predictable results. But predictability is a funny thing; you cannot build with confidence on a shifting foundation. The problem with conventional approaches is that they assume the foundation is firm; they have little tolerance for change.

The paradox is that trying too hard to create predictability creates opposite effect. Conventional practices are fragile in the face of change, and even in the face of learning. And yet, the more complex the system, the more necessary learning becomes. What is needed is an approach that encourages learning, and does not commit until learning is complete.

It should be obvious that decreasing the amount of speculation involved in making a decision increases predictability of the outcome. If you can make decisions based on facts rather than forecasts, you get results that are more predictable. Lean development is the art and discipline of basing commitments on facts rather than forecasts. It starts earlier, encourages change, freezes decisions later, and delivers faster than traditional practices, but nevertheless lean development produces outcomes that are more predictable.

The paradox of lean development is that you have give up some of the trappings of predictability in order to get true predictability. You have to abandon some conventional wisdom to gain the benefits of making decisions with more certainty. Fundamentally, you have to develop the capability to respond to events as they unfold, rather than hold dear the capability to orchestrate events in advance.

Principles of Lean Software Development

This research report discusses how lean thinking is applied in a software development environment. Software development is a broad subject, and no set of practices will apply to all software development environments. However, there are some fundamental principles that do apply to all software development environments. These principles are guidelines to aid in the formulation of practices appropriate for individual software development environments.

We've already mentioned four principles of software development:

1. **Start Early.** Start every development activity just as soon enough information exists to get started. Don't wait for the details; get everyone involved in figuring them out together. When development involves throwing information over a wall from one group to the next, none of the tacit knowledge critical to the success of the project will make it over the wall, and none of the essential feedback makes it back. Don't build any walls. Create high bandwidth, two-way communication flows among all participants by starting the learning cycles as early as possible.

2. **Learn Constantly.** Start with a breadth-first approach, exploring multiple options, but at the same time, develop complete, tested increments of functionality. Although these increments should production-capable, they are not 'final.' In particular, early increments are expected to change as the system emerges, so they require a simplicity and robustness that allows them to be refined as the details of the system emerge.

3. **Delay Commitment.** Encapsulation and loose coupling are the key mechanisms for delaying commitment. Although these techniques have been known for many years, object-oriented design brought them to the forefront of software development. To these we add refactoring (improving design as code is developed) and automated testing, which are essential for keeping code changeable not only during development, but throughout its lifetime.

4. **Deliver Fast.** The ability to deliver fast is the mark of excellent operational capability. The whole idea of delaying commitment is to make every decision as late as possible, allowing you to make decisions based on the most current knowledge. It makes no sense to delay commitment if you can't deliver fast. Speed decreases the length of the feedback loop and means you are acting on the most current information possible.

Alone, these principles are not enough. Four additional focal points are fundamental to lean development:

5. **Eliminate Waste.** The only thing worth doing is delivering value to customers. Anything else is waste. Seeing and eliminating waste is the first step to a lean value stream. Making value flow rapidly from receipt of a request to delivery is a fundamental principle of lean thinking.

6. **Empower the Team.** When the flow of work is rapid and responsive, there is no time for central control. The work environment should be structured so work and workers are self-directing. People, not systems, develop software.

7. **Build Integrity In.** Lean development produces a product with integrity – when flow is rapid, there simply is no room for shoddy work. This means that a complete test suite that documents the intentions of developers and the requirements of customers is part of developing software. Tests are not after-the-fact events, but are integrated into software development, controlled just as any other code, and become part of the delivered product.

8. **Avoid Sub-Optimization.** We have a strong inclination to break a complex whole into parts that are more manageable and manage each separately. However, this disaggregation has a well-documented tendency to create sub-optimized behavior. If you look behind self-defeating organizational behavior, you will usually find that it is caused not by the incompetence of people, but by measurements and expectations that optimize a part at the expense of the whole.

This report examines each of these principles in more detail, and discusses particular software development practices that can be used to implement them.

Predictable Outcomes

Wall Street has little sympathy for companies that can't meet their forecasts every quarter. In turn, senior management expects department managers to make and meet forecasts. By the time these expectations arrive at an IT department, meeting forecasts often becomes a significant challenge. Unfortunately, it seems that a large fraction of software projects fail to deliver on their promises for one reason or another.[1]

Why is it that software development projects seems to have so much difficulty delivering predictable outcomes? It seems that all too often projects are late or over budget or they deliver the wrong system or all of the above. How can the predictability of software development outcomes be increased? This executive report discusses a dilemma: in our zeal to improve the reliability of software development, we have institutionalized practices that decrease, rather than increase, the predictability of outcomes.

If this claim leaves you skeptical, consider that a decade ago, Japanese automakers routinely developed new model cars in 1/3rd less time, for half the cost of Detroit automakers. Only one sixth of the Japanese development programs were late, compared one-half of the US programs.[2] Yet the automobiles developed through these faster, cheaper programs were very successful. For example, Toyota's profitability from 1982 to 1998 was 1/3rd higher than Chrysler's, twice that of Ford, and three times that of General Motors.[3]

If you look at traditional software development practices today and automotive development practices a decade ago, you will find strong similarities. At General Motors, for example, a special team defined a *four phase process* which instructed each department what to do, when to do it, what results to produce, and where to send them. This extensive process was almost never followed in the real world of complex vehicle development, and did little to shorten development times or bring the other benefits expected of thorough planning. In fact, the more companies

attempted to define the process of product development, the less the organization was able to carry out that process.[4]

Japanese automakers use concurrent engineering to develop products. Light on process and heavy on communication, concurrent engineering has been widely adopted as the preferred vehicle development approach in recent years. Toyota refines concurrent engineering with *set-based design*; they start development early with a broad array of possibilities, and narrow the specifications through a series of increasingly accurate prototypes.

The paradox of set-based design is that development starts as soon as a vehicle concept is approved. It proceeds at an aggressive pace, even while multiple options are explored. And yet, commitment is delayed as long as possible. At Toyota, hard vehicle dimensions are not frozen until after the first parts are formed and bolted together – incredibly late by Detroit standards. Finally, once decisions are reached, scale up is very rapid.

The Predictability Paradox

The best way to achieve predictable software development outcomes is to start early, learn constantly, commit late, and deliver fast. This may seem to cut against the grain of conventional project management practice, which is supposed to give more managed, predictable results. But predictability is a funny thing; you cannot build with confidence on a shifting foundation. The problem with conventional approaches is that they assume the foundation is firm; they have little tolerance for change.

The paradox is that trying too hard to create predictability creates opposite effect. Conventional practices are fragile in the face of change, and even in the face of learning. And yet, the more complex the system, the more necessary learning becomes. What is needed is an approach that encourages learning, and does not commit until learning is complete. That is why Toyota does not attempt to tell die cutters the exact dimensions of a sheet metal stamping die until they have actually stamped out parts and bolted them together. Only then do they know for sure where the millimeters need to be shaved or built up to achieve a perfect fit. Other

automakers need to adjust the dies at the last minute also; they just pay a lot more for the 'unexpected' changes, or live with ill-fitting parts.

Michael Dell learned about not basing decisions on forecasts the hard way. At one point his young company bought a ton of memory chips, only to have them quickly turn obsolete, resulting in a multi-million dollar write-off. Dell decided that holding inventory was about the most risky thing his company could do. He decided that from then on he would respond to the market rather than try to predict what it would do. The rest is history.

It should be obvious that decreasing the amount of speculation involved in making a decision increases predictability of the outcome. If you can make decisions based on facts rather than forecasts, you get results that are more predictable. Lean development is the art and discipline of basing commitments on facts rather than forecasts. It starts earlier, encourages change, freezes decisions later, and delivers faster than traditional practices, but nevertheless lean development produces outcomes that are more predictable.

The paradox of lean development is that you have give up some of the trappings of predictability in order to get true predictability. You have to abandon some conventional wisdom to gain the benefits of making decisions with more certainty. Fundamentally, you have to develop the capability to respond to events as they unfold, rather than hold dear the capability to orchestrate events in advance.

Think about it this way. Throughout the 20th century, millions of people in dozens of countries bet that a planned economy would perform better than a market economy, and they lost the bet. Today, thousands of people in hundreds of companies are betting that a planned approach to software development will work better than a adaptive one. The extent to which they are right will no doubt be directly correlated to the capability of the domain to remain unchanged over the timeframe of the planned development.

Principles of Lean Software Development

For the remainder of this research report, we will discuss how lean thinking is applied in a software development environment. First a disclaimer: software development is a broad subject, and no set of practices will apply to all software development environments. However, there are some fundamental principles that do apply to all software development environments. These principles are not immediately actionable; they are guidelines to aid in the formulation of practices appropriate for individual software development environments.

We've already mentioned four principles of software development:

9. **Start Early.** Start every development activity just as soon enough information exists to get started. Don't wait for the details; get everyone involved in figuring them out together. When development involves throwing information over a wall from one group to the next, none of the tacit knowledge critical to the success of the project will make it over the wall, and none of the essential feedback makes it back. Don't build any walls. Create high bandwidth, two-way communication flows among all participants by starting the learning cycles as early as possible.

10. **Learn Constantly.** Start with a breadth-first approach, exploring multiple options, but at the same time, develop complete, tested increments of functionality. Although these increments should production-capable, they are not 'final.' In particular, early increments are expected to change as the system emerges, so they require a simplicity and robustness that allows them to be refined as the details of the system emerge.

11. **Delay Commitment.** Encapsulation and loose coupling are the key mechanisms for delaying commitment. Although these techniques have been known for many years, object-oriented design brought them to the forefront of software development. To these we add refactoring (improving design as code is developed) and automated testing, which are essential for keeping code

changeable not only during development, but throughout its lifetime.

12. **Deliver Fast.** The ability to deliver fast is the mark of excellent operational capability. The whole idea of delaying commitment is to make every decision as late as possible, allowing you to make decisions based on the most current knowledge. It makes no sense to delay commitment if you can't deliver fast. Speed decreases the length of the feedback loop and means you are acting on the most current information possible.

Alone, these principles are not enough. Four additional focal points are fundamental to lean development:

13. **Eliminate Waste.** The only thing worth doing is delivering value to customers. Anything else is waste. Seeing and eliminating waste is the first step to a lean value stream. Making value flow rapidly from receipt of a request to delivery is a fundamental principle of lean thinking.

14. **Empower the Team.**   When the flow of work is rapid and responsive, there is no time for central control.   The work environment should be structured so work and workers are self-directing. People, not systems, develop software.

15. **Build Integrity In.** Lean development produces a product with integrity – when flow is rapid, there simply is no room for shoddy work. This means that a complete test suite that documents the intentions of developers and the requirements of customers is part of developing software. Tests are not after-the-fact events, but are integrated into software development, controlled just as any other code, and become part of the delivered product.

16. **Avoid Sub-Optimization.** We have a strong inclination to break a complex whole into parts that are more manageable and manage each separately.   However, this disaggregation has a well-documented tendency to create sub-optimized behavior. If you look behind self-defeating organizational behavior, you will

> usually find that it is caused not by the incompetence of people, but by measurements and expectations that optimize a part at the expense of the whole.

This report examines each of these principles in more detail, and discusses particular software development practices that can be used to implement them.

Principle 1: Start Early

There are many software development failures that can be attributed to rushing in to development without understanding customer requirements or starting with a poor architecture which quickly calcifies into unmanageable code. Poor development habits such as lack of testing and source code management are indicators of impending disaster. In the face of these known failure modes, how can we possibly suggest that starting early is a good idea?

If time were really the cause of these problems and more time could really cure them, then certainly, we would recommend taking all of the time necessary to avoid software development disasters. But if time is a scapegoat for the real problem, then taking more time will not cure the problem. In fact, by ignoring the root causes of the software development problems we only give them the space to grow worse.

Requirements

So let's start by agreeing that understanding customer requirements is fundamental to successful software development. Gathering all the requirements at the beginning of development, however, is not the way to solve this problem. First, as we all know, users rarely have the ability to cleanly articulate what they really need. Secondly, users needs will change once they see the potential of the system and understand what it can do for them. Third, both the technology and the domain will change, and the longer the development time, the more it will change.

Worse, detailed requirements gathering has turned into a separate discipline, removing the people who gather requirements from the technical people developing the system. If there is one clear lesson we have learned from concurrent development, it is that the technical people making the day-to-day development tradeoffs must have a deep and intuitive grasp of what customers will really want once the system is delivered. This does not come from detailed requirements documents; it comes from excellent information flow between those who understand the customers and those who are developing the system.[5]

When gathering requirements is an arduous process with the results transmitted by paper to the development team, there is a weak, one directional, time-delayed information flow from customers to developers. The more time this takes, the weaker the link becomes. When developers are working on a complex product in an evolving environment, the bulk of the design will be determined by detailed trade-offs being made daily by the technical people working on the project. Doing this right requires a short feedback loop between the customers and the developers. An early start sets up this necessary communication flow early on. A late start gets in the way of this information flow.

Architecture

There are plenty of war stories about poor architectures that missed a key customer requirement like security or response time or some other critical feature that in the end resulted in a need to scrap the system and start over at a huge cost. So cautious managers believe that if design that is more detailed goes into a system before development starts, these disasters will be prevented. Unfortunately, this approach has a tendency to lead to premature design commitment, the very cause of the problem it is supposed to address.

Most people have a tendency to deal with complex problems by disaggregating the problem into its parts and focusing on the individual parts of the system. The problem with this approach is that the broad field in which the problem exists will not be examined closely once effort

moves to the sub-problems. Disaggregation means that the spaces between the parts and the interaction of the parts are easily forgotten.

Disaggregation is like creating many tunnels, each of which is explored in depth. But when you want to discover what you have not though of, what critical issue you have overlooked, it will probably not be found in one of the tunnels. Worse, once the tunnels are built, going back to the surface to dig a new interconnecting tunnel can be very painful.

Lean development takes a funnel approach, exploring the breadth of the landscape and only gradually narrowing the field of view. For example, Toyota drives new vehicle development through a series of prototype milestones. At the first milestones, several rough prototypes with broad tolerances are produced. As milestones progress, the number of prototypes and the tolerances are reduced, although two or three options are maintained and tolerances are not finalized until just before production.

The architecture of a complex system is not a static thing that can be designed at the beginning and left alone. The details of a good architecture emerge as skilled developers from all disciplines explore the problem together and refine the solution. Excellent product development performance requires excellent, detailed, bi-directional information flow among everyone on the development team:[6] analysts, architects, customers and customer proxies, database administrators, developers, help desk staff, maintenance programmers, technical writers, testers, typical users, user interface designers; everyone.

It is not time that leads to good architectures, nor more early detail; it is information flow among skilled people. Taking more time does not produce better designs; the skill of the development team and flow of information are the deciding factors. If you focus on getting the right people working together, they will take the system where it needs to go.

Discipline

Starting early and moving fast is not related to doing sloppy work. All great organizations have a culture of discipline, but this does not come from hierarchy or bureaucracy, it comes from disciplined people and disciplined thought.[7] In software development, discipline means that the basics are in place: source code control, version tracking, a build process, a testing process, coding standards, database and user interface policies, security screening, and so on. If these are not in place, you are not ready to get started.

It is not possible to move quickly or produce a quality product in an undisciplined environment. In the past several years, the measure of discipline of a software organization has been called its level of 'maturity,' and a long list of practices has been attached to each level of 'maturity.' However, the maturity model tends to focus on implementing a long list of processes, rather than on fostering the inherent discipline found in skilled, dedicated people.

Let's look at one example to illustrate this point. Several years ago a small company called Zeos, which assembled PC's in my home town of Minneapolis, won a Malcolm Baldrige award. This national quality award was a strong endorsement that the company's processes were very mature. At the same time, an equally small company in Austin Texas, which also assembled PC's, was spending its time figuring out what rapid flow of product directly to customers really meant. Instead of documenting its processes and submitting them to a maturity certification board, Dell Computer Corporation was investing in skilled people who invented new processes every week.

Disciplined software development habits come primarily from disciplined people who want to produce good results. No group of skilled developers cares to waste their time finding lost code or reconciling different versions. With wise leadership and appropriate training, good developers will gladly adopt the necessary practices and tools to enable them to generate high quality work in their particular environment. By all means, be sure to attend to this before trying to start early or deliver fast.

Principle 2: Learn Constantly

Constant learning seems like a good idea until you realize that it is the opposite of 'freezing' specifications. *Wait – slow down,* you say. *What is wrong with first planning the work and then working the plan?* Two things are wrong – if you don't know everything there is to know before you create a plan, then you are creating plans based on speculation. Thus, your outcomes will also be based on speculation. Secondly, things change. If you don't have feedback loops in your development process, you will be developing for the situation that existed at the beginning of the plan, not when the software is delivered.

Planning is a very good thing. It's the plans themselves that are generally useless, especially in software development. So do the planning, but throw out the plans. In lean software development, the goal is not to be able to forecast the future; the objective is to be positioned to respond to the future as it unfolds. This ability to adapt to reality is what gives all lean organizations their competitive advantage.

Iterative Development

The most fundamental technique for constant learning in software development is iterative development. Let's be very clear about what iterative development is all about. An iteration produces a small, tested, integrated increment of business value that is validated by customers and used as feedback for the next iteration. Iterations occur at short, regular intervals and they involve everyone: from architects to testers to the help desk staff.

Iterative development is the basic building block of lean development; it's the moral equivalent of 'Just-in-Time' in manufacturing. On the other hand, the method of implementing iterations will differ from one environment to another. So let's review the primary intent of iterations – and that is, learning.

Project management for a new vehicle model at Toyota means setting the dates for the regular prototype milestones: vehicle sketches, clay models, design structure plans, first prototype, second prototype, production trials, release to production. At each milestone, the tolerance level is reduced until there are only a few millimeters of tolerance in stamping dies, for instance, at production trials.

The process is sort of like carving an ice sculpture. You start with a large block of ice from which many things can be made. First, the big cuts are made, revealing the general shape of the sculpture. As time goes on, more detail is reveled by cutting away more options. Once options are removed, they are not revisited. This increasing refinement gradually narrows the options on the final appearance of the sculpture.

Similarly in software development, early iterations should leave many options open, but as time goes on, fewer degrees of freedom remain. Even so, early iterations should be complete – that is – they should be tested, integrated, even production-capable. However, they should not be 'frozen.' Early iterations represent one way – preferably the simplest way – that the increment of business value can be implemented. As development proceeds, the design is improved – or refactored – as new increments of business value are added, to keep it simple and remove any repetition. Early on, significant refactoring can be expected. In some domains – embedded systems come to mind – tolerance for change might narrow as the design funnel narrows. If this is the case, then the areas most likely to change should be the subject of early iterations.

Synchronization

When many people are involved in developing a complex system, there must be a way to synchronize the work of multiple teams regularly so their learning can be merged effectively. There are several ways to do this, depending on the domain. A standard approach when code is shared among developers is to have a regular (at least daily) code check-in process, followed by a build, after which an automated test suite is run. Often called the 'daily build and smoke test', this is an essential technique

for providing a short feedback loops in just about any development environment.

Another synchronization technique is to develop a simple *spanning application* that demonstrates basic functionality through all layers of the system. A spanning application has also been called a *tracer bullet*, *thread*, and *spike*. By whatever name, the idea is the same. Once the fundamental software approach is demonstrated, multiple teams can implement the same design across the broader system.

A third technique is the *matrix* approach, where individual teams are assigned responsibility for various modules. The project starts by developing the *cross-team* capabilities – the interfaces between the modules. Once the inter-team software is actually running (not just defined, actually running), the teams develop their assigned modules using iterations, continually integrating their increments of development into the overall framework, which was developed first. This technique develops the most difficult, risky, and communication-intensive portion first; leaving the relatively easier part until last.

Principle 3: Delay Commitment

Delaying commitment means keeping your options open as long as possible. The fundamental lean concept is to delay irreversible decisions until they can be made based on known events, rather than forecasts. Thus, Just-in-Time focuses on assembling the final product after an order is in hand. Even when decisions have to be made before all of the facts are in hand, there are many ways to delay commitment.

For example, suppose I want to plan an outdoor wedding in my home state of Minnesota on August 10th. I have to send invitations out many weeks in advance, and about the best I can do in forecasting the weather is count on a temperature somewhere between 65⁰ F and 95⁰ F. There is no way to know if it will rain at the time the invitations must be sent. In Minnesota, we deal with this uncertainty by renting a tent. As long as we have the tent, we erect the top in any case. If it's sunny, the guests will probably

appreciate the shade, and if it is rainy or cold, we can add sides to the tent at moment's notice.

Take a look at every decision you make, determine what kinds of forecasts the decision is based on, and the certainty of those forecasts. If the level of certainty is low, then the first thing to do is to try to delay the decision as long as possible. If the decision can't be delayed, the next thing to do is to 'rent a tent,' that is, find ways to reduce the dependency of the decision on the forecast.

The Last Responsible Moment

Decisions should be make at the last responsible moment: the moment at which failing to make a decision eliminates an important alternative. Making decisions before the last responsible moment means making them without the best possible information, but delaying a decision beyond this point means letting the decision make itself. Making decisions at the last responsible moment does not mean procrastinating so that decisions are made by default.

You need to develop a sense of when decisions must be made and then make them when their time has come. It is equally important to develop a sense of what is critically important in the domain and make sure these areas are not overlooked in the decision-making process. If security and response time are important in this domain, they must surface early so that when the time comes to make decisions in these areas, issues have been fully investigated and informed decisions can be made.

One of the easiest ways to delay commitment significantly is to emphasize the expectation that developers share partially complete design documentation. Usually there is a reluctance to do this; most people tend to want to do their job completely before involving downstream functions. However, developing a complete design before sharing it forces upstream departments to make commitments early, and encourages the *throw-it-over-the-wall* syndrome. It is far better to share partially complete designs and arrange for direct, cross-functional collaboration.

Decisions can be delayed much longer if you have developed a quick response capability. Just-in-Time assembly is the quick response capability that enables Dell to wait until an order is received before the computer is assembled, and yet ship the computer within a week. Without fast response, it is not possible to delay commitment until forecasts turn into orders.

Foster A Sense of How to Absorb Change

There is an art to delaying commitment in software development, and it involves developing a sense of how to absorb changes. When you are developing a large, complex system, you probably do not want to include every degree of freedom you can imagine. It is more practical to understand the domain well enough to understand the most likely axes of change. The idea is to encapsulate the areas most likely to change, and separate areas that will probably change independently. Object oriented design assists in this because thinking about the domain as objects tends to both encapsulate areas that are likely to change at the same time, and separate areas that are likely to change independently.

Virtually every enterprise system these days has a layered architecture, separating user interface, business rules, and database or persistence into separate layers. A layered architecture is a good start at encapsulation and separation of concerns as long as lower level layers do not depend on higher-level layers. Thus, the business rules should not depend on the kind of user interface that is being used, and there should not be any business logic in the user interface. Similarly, the business logic should not depend on what kind of database is being used, and business logic should not be implemented in the database.

A key technique for absorbing changes easily is to avoiding repetition like the plague. If you have to say the same thing in more than one place – either in design documents or code – then refactor the design to consolidate the capability into one place. One of the most effective ways to facilitate change is to localize every potential change in only one place.

An even more effective technique for absorbing change is to develop and maintain an automated test suite as part of the system. In a complex system you have no idea what the unintended consequences of a change might be, so the trick is to maintain a current test suite which allows changes to be made with confidence. Automated tests will be covered in detail in the section on integrity.

Write Less Code

An obvious technique for delaying commitment is to write no code before its time. This means avoid developing any extra features just because they seem like they would be 'nice-to-have.' Implementing features that are not needed, on the speculation that they might be useful, adds a lot of long-term baggage: source code tracking, testing, documentation, training, help desk support, maintenance, and on and on.

In addition to adding excess baggage, implanting capabilities before they are needed is simply a bad idea. Why? First, it is speculation to assume that the features will actually be needed in the future exactly as you implement them now. There is a good chance that things will change and your work will have been for naught. Worse, an early, not quite correct implementation tends to interfere with the correct implementation later on. The excess baggage that was added is hard to ferret out and fix.

But there is an even bigger problem with early specification and implementation of features. In one investigation by the Standish Group, it was found that 45% of the features and functions of a system were never used, while only 20% were used frequently.[8] Few people in software development dispute these numbers. Most find that it meshes with their experience that close to 2/3^{rds} of the features in a typical system may be rarely or never used.

If we could eliminate the 2/3^{rds} of the features in a typical system that are simply excess baggage, we could write systems with only 1/3rd of the code and even a great reduction in complexity. We would have far less documentation, testing, support, and opportunity for failure. This would indeed be a dramatic improvement in software development.

How do we get rid of all those unnecessary features? For one thing, we stop asking people for a laundry list of desirable features at the beginning of a software development project. Instead, we start by implementing top priority features first. We resist the temptation to add extra features, even those further on down the priority list, until they become the top priority. We get the simple, basic functionality working and let the customer discover before we implement them that most of the features on their wish list are things they won't use anyway. This lean approach to managing scope is the fastest, easiest way to write less code. It will be discussed further in the section on eliminating waste.

Principle 4: Deliver Fast

To those who equate rapid software development with hacking, there seems to be no reason to deliver results fast, and every reason to be slow and careful. But if you look beyond software development, consistent fast delivery is an indicator of superior performance. When you send an overnight package via Airborne or Federal Express or UPS, you know that reliable next day delivery comes from excellent operations.

I can remember when I used to fill out a form and mail it into Sears, and receive my order in a couple of weeks. Then LL Bean started taking orders by phone and shipping the next day. Suddenly Sears service, which had been adequate for decades, started feeling very slow. Unable to keep up with industry standards for speed of service, Sears eventually was forced to close down its century-old catalog store.

In most areas, speed is associated with superior quality. You prefer to shop at a store with short checkout lines. You want broadband Internet service. In fact, consistent fast delivery is not possible without high quality. Just-in-Time manufacturing is not possible when the production process cannot be depended upon to produce high quality product.

Returning to software development, think of reliable, repeatable, rapid delivery as the sign of excellence in a software development organization. We're not talking about rush jobs, we're talking about a sustained capability to deliver working software quickly. If your organization can

do that, you probably don't have to measure how mature it is, you know it's right up there with the best.

Queueing Theory

When you get stuck in a traffic jam, it's no comfort to know that traffic engineers study the mathematical theory behind your predicament. But the next time you are stuck in traffic, you might spend the time thinking about traffic jams in your organization. What causes traffic jams? Too many cars for the carrying capacity of the road. Does work flow seamlessly through your organization, or is there more work flowing through your organization than there is capacity to handle it? Have you evaluated the overall impact of slow movement of work through your organization on the company?

Perhaps you hate to see anyone in your organization standing around with nothing to do, but when the tables are turned and you go to a grocery store or stand in line at the airport, you probably wish the store or airline had a few people standing around just waiting to help you. Queueing theory is the study of how to achieve both goals – keep the waiting lines as short as possible and the work flowing as fast as possible while providing best utilization of restricted resources.

The paradox in queueing theory is that you do not get the best utilization of scarce resources by running them at 100% capacity. You already know this if you run a computer operations center; you would never run the servers in your computer room at full capacity, because you know traffic slows to a crawl when you do. Have you considered that the same thing probably happens when your people are working at better than full capacity?

Queueing theory demonstrates that reducing the size of batches moving through a system allows you to provide faster service at higher utilization levels. The idea is to divide development up into many small batches of work that flow through your organization, rather than a large lump of work that moves *en masse*. In the last section we suggested that early release of partially complete design information is a good way to delay

commitment. We further suggested developing systems in small increments. Here we see that these practices are also good ways to improve the flow of work through an organization and increase delivery speed.

Self-Directing Work

It never ceases to amaze me that I can sit at the computer in my home office, print out an Airborne label, and schedule a pickup. About 5 or 6 pm someone rings my doorbell to pick up the package. At 9:00 the next morning, it's delivered half way across the country. As my package moves on its way, quite a few people have to do the right thing to get my package delivered on time. I often wonder, how do all of those people know what to do?

The secret is that every package has directions on it. Managers do not try to schedule each person's day in detail; they set up a system where the package labels tell drivers and sorters what to do. You will find the same thing in any fast-moving situation. When emergency workers arrive at an accident, the situation and their training tell them what to do. Just-in-Time manufacturing works just like stocking shelves at a supermarket: when a shelf is emptied of inventory, the feeding workstation makes whatever is needed to restock the shelf.

In all rapidly moving operations, management's role is to set up a self-scheduling system, balance capacity, and train workers. Then the system itself, and fellow workers, send signals that indicate what is to be done. You will not find a fast-moving system that uses central scheduling, because there isn't time for a central system to evaluate all of the options, deal with all of the variances, and make correct decisions. People are much better at that.

Let's take another look at Toyota's product development scheduling mechanism. The chief engineer of a new model sets the dates for the regular prototype milestones: vehicle sketches, clay models, design structure plans, first prototype, second prototype, production trials, release to production. There are no master schedules or Pert charts or Gantt charts

or earned value tracking. The people in each function know what is expected of them at each milestone, and they deliver. It's that simple. If engineers need information or subassemblies in order to meet their deadline, they are expected to get them. There are no excuses; everyone figures out for themselves how to meet the deadlines.

A rapidly moving schedule is best implemented by intelligent people, not centralized planning. The management challenge is to organize work so that the people have the training, tools, capacity and motivation to deliver reliable, repeatable results. When your organization can do that, you have a mature organization.

In software development, the moral equivalents of Toyota's prototype milestones are regular iteration deadlines. Early on, the overall iteration plan is sketched out, and then at the beginning of each iteration, the exact goals of the iteration are fleshed out. The iteration planning meeting is the place where customers or customer proxies set priorities, and the development team estimates it's capacity for the next iteration. If the iteration is a month or less, it is reasonable to expect that the immediate business priorities can be established and development time can be accurately estimated.

Thus the iteration planning meeting establishes the expectations for the next milestone. If the development team can reliably produce working software delivering the expected increment of business value by the deadline, then the organization is working well. If the expected business value tracks the overall iteration plan that was sketched out at the beginning of the project, then the project has a high degree of predictability. If it does not track the original plan, then there is most likely something wrong with the overall plan.

Self-directing work is fast, reliable, disciplined, and a very accurate predictor of the capability of the organization. It provides quick, accurate feedback to management, to the business, and to the development team. Some managers have difficulty believing that by turning detailed direction over to the people involved, they will get better, faster, more predictable results. Yet in any system in which the underlying pace of change

outstrips the planning cycle, self-directing is the only practical way to deliver reliable, repeatable results.

Principle 5: Eliminate Waste

The first principle behind lean thinking is to focus all efforts on adding value for customers, and to make the value-adding activity flow as rapidly as possible. In practice this means selecting a few end-to-end customer processes and focusing on how much value is created and how fast value is created in each process.

Walk the Value Stream

A good exercise for eliminating waste is to follow a customer request from the time it arrives at your organization until the time the customer is satisfied. Walk in the shoes of a customer as their request comes into your organization, goes into your priority setting mechanism, comes out approved, arrives at your development organization, makes its way through your development process, is installed at the customer site, and the expected business value is (or is not) delivered. Firmly planted in your customer's shoes, visualize each step of this process, and think about how you might deliver more value, more reliably, and faster.

First look for bottlenecks. How long does the approval process take (on average)? Once development starts, does it keep moving at a reliable pace until it's done, or are there bottlenecks which put development on hold while, for instance, you wait for a design review or testing? Why aren't these things integrated into the development flow? Are people multi-tasking and slowing down all the projects? Are dates for the next milestone known and reliability met? How long does it take to deploy an average system to production? How much training does your customer's staff need?

As long as you are in your customer's shoes, forget for a moment about your organization's goals and think about your customer's goals. Is their domain evolving? Do they need change tolerance? Do they have many people using their software? Do they need high usability? What kind of

growth are they expecting? Do they need extensibility? How vulnerable are they to intrusion? Do they need extra security? Do your end-to-end processes discover and support these goals? And finally, who in your organization cares about these customer issues?

Seeing Waste

Waste is anything that does not add value, as perceived by customers. In lean thinking, eliminating waste is *the* key lever for achieving excellence. But before you can eliminate waste, you have to recognize it for what it is. You might think, for example, that your change authorization system is essential to control scope creep. But there are two things wrong with this assumption. First, there are better ways to control scope (more on this later), and second, the change authorization system you are using is probably not perceived by your customers as adding value. After all, your system is probably there to keep your customers from changing their minds. Why would they find that valuable?

Let's take a deeper look at waste. From a customer perspective, any time spent waiting to achieve the perceived business value of their request is wasted time. The time their request spends sitting in your in-box waiting to get scheduled into your organization is waste. So is the time the request spends waiting in other queues in your development process. Even though it may not be your problem, the time it takes to deploy the system to production is waste. One thing is for sure, unless these bottlenecks are recognized and addressed, they are not going to go away, and the waste will continue.

In lean manufacturing and lean logistics, inventory is considered waste. This doesn't mean you have to eliminate all inventory, but the idea is to keep it at a minimum, because it adds no value. Not only is inventory kept low at a manufacturing plant or warehouse, but throughout the value stream. Thus not only will Dell have low inventory at its plants, but its vendors will also keep very low inventory as well. Dell does not achieve its rapid delivery capability by forcing suppliers to take risks with inventory, it focuses on minimizing the risk of inventory across the value

chain. Moreover, because Dell delivers very quickly, its customers can keep inventory low also.

The inventory of software development is partially done work. So as you look for waste to eliminate look at the queues of partially done work in your organization. Actually, queueing theory indicates that the level or partially done work is directly proportional to the speed at which an average request moves through your organization, so if it takes a long time for you to respond to customer requests, you know you have a lot of inventory of partially done work. The objective is to find this waste and eliminate it. You can measure whether or not you are successful either by measuring the speed of your end-to-end processes or by measuring the size of the queues of unfinished work.

Doing extra work is also waste. Do you really need a user's manual, or would it be better to focus on making the system so usable that a manual is unnecessary? A lot of help desk time is wasted when people have to look things up in a users manual. In fact, one company attributed a million dollar drain on its profits to a printer driver that was too difficult for users to figure out, so they swamped the help desk until the software was recalled. You might kill two birds with one stone and eliminate the user manuals while making life easier for your customer's support people by focusing on usability.

Task switching is waste. If you have two projects, each of which takes two months, can you get both done in four months? Probably not, because your people will be changing context frequently, which takes time in an activity such as software development where good ideas 'flow' only after uninterrupted time spent immersed in the details of the system. Moreover, many people will have dual commitments, and if priorities are not clear, some people will favor one project, some the other project, so each will be cause the other to be delayed. It is usually far more efficient to do one project first, and then the other.

How much time and motion does it take developers to get answers to questions? How much time do they loose waiting for answers from customers or other developers? If your software is going to flow rapidly

through your system, the waste of walking around looking for answers and the waste of waiting for other people has to be eliminated.

Some have thought that if everything could just be written down and handed off to the next person, there would not be so much time wasted waiting to find out answers, people could just refer to the documentation. But experience has shown that the opposite is true – handing off documentation *over the wall* to the next department causes waste. The tacit knowledge of the people writing the document doesn't make it into the document, and the creators of the document don't think of many details needed by the receiving department. Sequential hand-off of documentation is a big creator of waste.

Creating defects is waste in direct proportion to the length of time the defect is in the system. So if a big defect is created and discovered immediately through testing, it doesn't create much waste. If a small defect hangs around for a long time and is discovered just before shipment, it creates a much bigger waste. The idea is to catch defects early, through some combination of review and testing appropriate to the domain.

Scope

The biggest waste in software development are the features and functions that are developed, tested, and supported, yet rarely if ever used. Earlier we noted that perhaps $2/3^{rds}$ of the software in a system might fall into this category. Whatever the ratio, it is far too large, and none of the scope control mechanisms we have used so far has made a big dent in it.

Here's the problem. Suppose you say to customers, "Okay, we're going to give you a system to solve XYZ problem. We're going to gather all of the requirements, analyze them, and write them down in a big book for you to review. Your job is to make sure we have gotten everything right, because once we start coding, you can't make any changes. So we're going to ask you to sign-off on the requirements, and then if you want any changes, we'll estimate the cost and you'll have to approve the change.

Now let's assume that the customers don't really know what they need to solve XYZ problem (that's why they asked you to solve it). Thus, they are worried that they might forget to tell you something as you gather requirements, so they try to be very thorough. The process does not ask them to be sure that they need something when they put it on the requirements list; the process encourages them to put everything on the list unless they are certain they will not use it.

Is it any wonder that a system developed under these rules will have many features that end up in the rarely-if-ever-used category? In our standard approach to defining scope, we build in tremendous incentives for excessive scope. Freezing scope usually has exactly the opposite effect that we desire. It does not prevent scope creep; it causes scope bloat.

So what's the alternative? First, accept the fact that customers cannot clearly specify at a detailed level exactly what they want; in fact, most won't know what they want until they see it in production. So implement in small increments, in priority order based on business value, and get customer feedback after every increment. Encourage early release to production, or at least production trials, of skeleton systems with only high priority features. Let the customers discover what they really need and will actually use, and develop only those features. If possible, discard any features that are not used. By starting early and committing late, you can write far less code.

Let's acknowledge right at the start the term *scope* has a bit of a condescending ring to it. Customers don't care about scope; they care about business value. The people who worry about scope are those who are trying to keep customers expectations in line with what's reasonable (or what's funded). So scope is sort of a fence we put around customers to keep them in line. Customers get even by pushing on the fence so that it surrounds more territory.

We need to get rid of the fence and the implied condensation, and start using trust. Customers need to trust developers to deliver the required business value in the timeframe and for the cost that is justified by the value. Developers need to trust customers to clarify the priority of

features and let them stop developing when they run out of time or money or both. If both organizations are in the same company, establishing this trust should not be so difficult. When two or more companies are involved, trust is a whole different issue which we will discuss later in this report.

Principle 6: Empower The Team

Lean organizations are profoundly centered on the people who do the work. If this is true in manufacturing and logistics, it is far truer when knowledge workers are on the front line. Yet it seems to be difficult for many organizations to truly center on the people who develop software.

There are many people involved in software development. On the one extreme, we have the learn-on-the-job loner who can rightly be called a hacker. At the other extreme, we have the process police telling developers how to do their jobs. In the middle, we have many people who very much want to do a good job, but will not tolerate condensation.

The basics have not changed. You need to have the right people. You need to be able to train new people and foster expertise. You need wise leadership and cross-functional teamwork. You need the right mix of discipline and self-determination. It's all very easy to say. Getting there is another matter entirely.

If there is one guidance that lean thinking can offer in this maze, it is that people should design their own processes, and these processes should be expected to change continuously. This contradicts the prevailing emphasis on defined, documented processes, which software development theoretically adapted from the best manufacturing practices. I would argue that defined, documented processes do not constitute 'best practices' in manufacturing. Excellent manufacturing practices start and end with front line workers defining and constantly improving their own processes.

Manufacturing Process Design

In the 1980's an accidental experiment was set up that showed how much the expectations of management could influence both the skill and the motivation of front line workers. The General Motors Fremont, California plant was closed in 1982 to end a record of abysmal quality and a history of problems with a rancorous union. Two years later the plant re-opened as the New United Motor Manufacturing, Inc., or NUMMI, under Toyota management. The experiment began when Toyota was required to hire back 85% of the union members and all of the union leadership. With the same workforce and the same job, the only difference was management style. Would things really change?

Within two years, the productivity of the NUMMI plant surpassed that of all General Motors plants, and to this day, it remains one of the most productive plants in the country. The results of the experiment are clear: the same general group of people with different management turned in spectacularly superior results and sustained these results over the long haul. What was different?

Toyota management started out by avoiding the condensation inherent in the traditional approach to automotive manufacturing. Instead of hiring back the hundreds of industrial engineers who typically be needed to design the manufacturing jobs, they taught the workers how to be industrial engineers and design their own jobs. Aggressive training programs and strong peer pressure combined to sort out incompetent workers and encourage skilled workers to improve continually. The bottom line is, a management emphasis on front line workers defining and constantly improving their own processes accounts for most of the difference between abysmal performance and stellar results.

Nucor Steel, which went from nearly bankrupt in 1968 to the largest steel company in the United States today took the same approach to its workforce. It located its plants in farm communities and had local contractors build the plant. The contractors that did a good job could expect to be hired and work in the plant they helped build. Nucor hired

people with an excellent work ethic and superior mechanical ability, and they let them design new steel-making processes as well as their jobs.

A Management Process

In the late 9180's, after Jack Welsh rationalized GE's businesses and eliminated many people, he spent some time talking to front line workers. They told him that there was just as much work in the company as before, but far fewer people to do it. So Welsh talked to their managers, who agreed that there was unnecessary work. Welsh wanted to know why the managers didn't eliminate the extra work, but they complained that they were also swamped, and had little time to make changes. So in an effort to get unnecessary work out of the business, a frustrated Welsh invented something that he called Work-Out.

Of all the corporate programs over many years, the GE Work-Out has been one of the most uniformly successful and one of the most accepted by employees. This is probably because Work-Out is a process that tells managers what to do, rather than telling workers what to do. At a Work-Out, the senior manager hosts a meeting with a few dozen front line workers, who spend two days coming up with proposals on how to eliminate waste and make their work flow faster. The proposals are presented to the senior manager, who is required to say 'yes' or 'no' to each proposal on the spot. If the answer is yes, then the people who made the proposal are chartered to implement it immediately.

Software Process Design

Instead of looking for the perfect process to use to assure that your organization will develop great software, devise a way to enable the workers in your organization to design their own processes. Start with the assumption that every capable developer wants to work in a stable environment with good basic practices, and that your developers already know what is wrong with your environment and how it can be improved. Creating a superior software development environment involves unleashing the dedication and brainpower of the people doing the work.

The fundamental enabling mechanism for people to design their own process is to have a good understanding of the purpose of their activities. In the Marines, leaders are expected to communicate *end state* and *command intent*, rather than giving specific directions. Similarly, a development team needs to know what the group needs to achieve in order to design appropriate processes. A control system for a machine in a factory will require entirely different processes than a web page for filling out a survey. A system that falls under federal regulation has different parameters than a sales support system.

Once a team understands what they are supposed to accomplish, they will know if existing processes are up to the task, need beefing up, or are too overbearing. Every software development project should start with a team selection of processes appropriate to achieve the mission, and every iteration should start with a review and update of the processes used in the last iteration. If a team finds that their processes fall short and need improvement, they must be given the time to put the necessary structure in place to allow them to do a good job. If a team finds that corporate processes are inappropriate for their task, they must have the trust and freedom to use alternative processes to accomplish the same overall goals. Management's job is to establish a framework in which teams can use their collective brainpower to determine the best way to do their job.

Condescension

All too often, the actions of managers and central staff groups telegraph the message to workers: "*We're smarter than you.*" Before NUMMI, the GM Fremont plant had scores of industrial engineers going around with stopwatches designing the jobs of the workers. The message was clear: *We're smarter than you are*. One of the key reasons Toyota management was so successful is that they made the workers feel they were being *treated like adults*.

A few years back when experienced software developers were hard to find, there was a search for processes "designed by geniuses to be run by idiots,"[9] which would enable inexperienced and ill trained people to successfully develop software. Even if such processes could be found,

they simply could not compete against processes that leverage the intelligence and skill of developers. Worse, the fundamentally condescending tone of such processes encourages developers to disengage from the success of the project and the organization.

Many organizations have departments, often called *staff groups*, chartered to gather knowledge and educate the rest of the company in some area of expertise – for example, Capability Maturity Model (CMM) or some other good practice that management wants to see disseminated throughout the company. Unfortunately, knowledge gatherers and disseminators often approach their job by developing and enforcing standard processes. The message they send loud and clear is, "*We're smarter than you.*" Simply using the word *'maturity'* telegraphs a message of condensation.

Staff groups in a lean organization start with the assumption that development teams know their jobs and their problems better than anyone else does. The role of the staff groups is to make itself so useful that it is invited to help developers work on their problems; staff groups in a lean organization would not consider trying to impose software development processes by edict.

Leadership

In the book *Good to Great*,[10] Jim Collins discusses the distinguishing characteristics of an organization that moves from mediocre performance to sustained superior performance. You don't find strategy or process anywhere on the list of factors that move companies from good to great; but at the top of the list are these two characteristics: Level 5 leadership and getting the right people in the right positions.

A Level 5 leader is someone with a blend of personal humility and professional will who leads skilled people by quietly and effectively bringing out their best efforts. Level 5 leaders are found in the background, crediting other people when things go right, stepping forward only when things go wrong, to take the blame.

Companies that are excellent at product development usually place a Level 5 leader at the helm of a new product development effort, whose job it is to understand the customer requirements and constantly keep them in front of the developers. Toyota, for example, has a chief engineer in charge of every vehicle development program. A chief engineer has ultimate responsibility for understanding the customer, developing the vehicle concept, and transmitting that concept regularly to the engineers making the day-to-day tradeoffs that will eventually determine the success of the car. Sometimes the chief engineer has been called a heavyweight project manager, but this is not a good characterization. The chief engineer at Toyota is a Level 5 leader, someone who knows how to get things done through others without heavyweight tactics. A chief engineer does little in the line of management, but rather focuses on the technical architecture of the automotive system and the overall business success of the car.

Complex software development efforts require Level 5 leaders who combine deep understanding of the domain with the technical knowledge needed to understand the day-to-day decisions that developers must make, and facilitate a broad information flow between the two. Just as a chief engineer has the technical competence to oversee the system design of an automobile, complex software development needs a technical leader capable of overseeing the architecture of the system and assuring that this architecture will meet the customer needs.

Excellent designs concepts often originate with an experienced technical leader, but they are never the work of one person. An effective software development leader must be able to marshal the forces of the entire development team, while developing a deep understanding of the customer domain. Architects who are removed from the day-to-day development, or those who are not deeply familiar with the customer domain, are unlikely to design great products. Worse, architectural edicts from staff groups usually come across as uninformed and condescending to a development team.

For smaller projects, it is usually a good idea to let leadership emerge rather than trying to appoint a leader. Level 5 leaders tend to work in the

background to create an effective environment. Left to its own devices, a team will find and follow such a leader; in fact, various leaders often emerge at different points of a project. Management can rarely select a better leader than the one a team selects for itself, and encouraging the emergence of leaders in small teams is one of the best ways to find and develop the Level 5 leaders needed for larger projects.

Expertise

The second characteristic of *Good to Great* companies is that the Level 5 leader starts by getting the right people on board and in the right positions and the wrong people off doing other things. Then, with the right expertise in place, the team figures out where to go and how to get there.

No amount of process can substitute for expertise. If you are converting an Oracle database to a SQL database and the developers have never programmed stored procedures before, you have a recipe for disaster. If you are designing a web site with developers who have no background in user interface design, you can't expect the web site to have the usability characteristics that facilitate high sales. If you are developing new business rules with developers who have no background in object-oriented thinking, you should not expect wise encapsulation nor appropriate separation of concerns.

Certainly everyone does not have to be an expert, but you must have appropriate expertise in all critical areas of the system either on the development team or readily accessible when needed. You should also have a good method for transferring expertise broadly within a team, so that no one person becomes indispensable. Pair programming has proven enormously effective at facilitating such knowledge transfer; well-structured design reviews also work well.

In general, every area of specialization that constitutes a competitive advantage should be supported by a center of expertise that fosters learning and knowledge transfer in the specialized area. Of course, just like staff groups, these centers of expertise are teaching and support centers. If they begin to transmit the message "*We're smarter than you*,"

rather than "*We're here to support you*," they will rapidly loose their effectiveness.

Principle 7: Build Integrity In

Lean organizations always produce high quality products – there is no way they can consistently commit late and deliver fast if they have sloppy procedures or make a poor quality product. Since lean organizations focus on delivering value to customers, they develop a deep understanding of what value means, and constantly tune their organization to spend all of its time creating customer value.

Rethinking Testing

Traditional software development practices focus on requirements gathering and traceability as the foundation of a quality product. Lean software development focuses instead on testing as the foundation of product integrity. Lean thinking starts with the assumption that all people, at some time, will make mistakes, and thus it is necessary to mistake-proof all human activity. The idea is not to train and exhort people to do a better job. The idea is to assume that every mistake that can possibly be made will be made, and so mechanisms must be put into place to make mistakes impossible.

I remember adding disk drives to PC's some years ago, and more likely than not things would not work when I turned on the power. After a while it was automatic – I would check the IDE cable and sure enough, I had put it in wrong. Virtually all other the cables in a PC can only be put in one way – you can't put them in upside down or backward. But the IDE cable was different – not only could it be put in upside down, it could easily be shifted one or two pins left or right of the correct position. Yet it was very difficult to detect these mistakes visually. I'm sure that the designers assumed that only 'experts' would add new disk drives to computers so they did not need to mistake-proof the cable. Well, I was a process control engineer, as expert as they come, but I still got the cable plugged in wrong many a time. Today IDE cables have one plugged hole and one missing

connecting pin, so it is impossible to put it in the wrong way. It took the industry far too long to fix this problem.

Software systems should be designed to be mistake-proof. Any software system that might be changed in the future should come with a complete set of automated tests, so that when changes are made, it will impossible to break the rest of the system. Systems that are delivered without the associated tests are like the IDE cable of old. They are missing the obvious mistake-proofing feature that should be part of any system. Test suites are not just for novices – in fact, just as with the IDE cable – experts probably need test suites more than anyone does.

Tests are not an after-the fact event used to check whether the developers did things right. Tests are used at every step of development to see if the intention of the developer was properly implemented. Virtually all developers test their code once it is written to see if it really does what they expect it to do. Unfortunately, these tests are usually informal and are rarely captured.

Since I was a process control engineer, I wrote code that moved equipment, and the equipment was not in my office as I wrote the code. Usually it was a long airplane ride away. So I always wrote a simulator for my code, and as I added each new feature, I ran it through the simulator. With rare exceptions, I could count on my code working correctly the first time I hooked it up to hardware, and my reward was a faster return home to my family.

I developed used multiple techniques to mistake-proof code. Often I needed to write assembly language code to optimize performance. I would first write the algorithm in a high level language, prove out the logic through test sequences, hand-optimize the assembly language version of the high level language, and re-run the tests. Since logic verification and performance optimization are different things, I mistake-proofed the logic before optimizing for performance.

Techniques such as these which incorporate testing and integration into the development process are fundamental to producing a product with

integrity. Writing code without testing it thoroughly strikes me as a poor practice, one which will extend the time necessary to produce a working system. As long as tests will be done during development, it seems only logical that these will become part of the delivered product, maintained as part of the code, changed as the system changes, and run every time there is a new build. Tracing code back to requirements, sending code off to a separate testing group, and documenting the system, all strike me as practices that distract developers from using the most reliable method of assuring that software works and is maintainable: tests.

If you have limited resources and can choose only one approach to developing a sound product, then including an automated test suite as part of the deliverable code is by far your best investment. Although tests alone are not likely to constitute a complete description of the system requirements, they are the best way to document the detailed understandings reached between developers and customers as the code is developed. Even when a final run through a test group is standard practice, it is no substitute for having testers involved in the day-to-day development work, providing immediate feedback to developers and helping document through tests what users really want. Although some final documentation is a good idea, those who maintain the code are unlikely to trust it, but they will trust a working test suite to help them find unintended consequences of their changes.

An automated test suite makes concurrent software development possible. You cannot start early and decide late if you cannot make adjustments with confidence along the way. The whole point of concurrent development is to develop a capacity to change so commitments can be delayed as long as possible. An automated test suite is the key to developing a capacity to change. With so many benefits, it's amazing that every system isn't routinely developed and delivered with an automated test suite as an expected part of the deliverable.

Developing by Feature

In the past, software tests were categorized into unit tests, system tests, and integration tests. This hierarchy of tests is left over from the days

when individuals 'owned' individual software modules. However, these days we do not program software module-by-module, we program software feature-by-feature. This generally means that the code in any individual module is commonly owned, so that when a developer adds a feature, she can make changes to all the modules necessary to implement that feature. When the next developer makes changes to the same modules, he has to make sure that his changes are compatible. For this reason, we need tests. But these tests are not limited to individual modules, because when we develop by feature, we generally integrate new code into the overall system right from the start. Thus, everything a developer does should be tested at the unit, system, and integration level just as soon as possible.

It is a good thing that we are no longer building systems module by module, then trying to integrate everything at the end of a project. When integration tests were left until the end, and they were notoriously difficult to pass and often required a lot of redesign of individual modules. By moving integration activity much further forward in the development process, we not only solve the tough problems early, we get outcomes that are more predictable.

So, if we no longer have unit, system, and integration tests, what types of tests do we have? The most basic type of test is the one we already discussed, the developer test. These tests should be written by the developers to test that the mechanisms they intended to implement actually work. They should be automated and run after every build. Developers who are merging in new code before they actually check the code into the source control system might also use a private set of tests.

There must also be customer tests, which test whether or not code addresses the customer needs. Although some approaches suggest that customers write customer tests, in practice testers or analysts who represent the customers usually write them. Generally, customer tests should be automated, but sometimes they will start out as manual tests. Testers might also recommend exploratory tests and other methods to test that features work as customers would expect.

Another category of testing which is important for insuring integrity is usability testing. The idea behind usability tests is to give users who know nothing about the system a set of tasks and watch quietly as they work their way through the tasks. If the system is not self-explanatory or if they make mistakes, then the system gets a lower usability score. Usability can make tremendous differences for on-line storefronts, where usable web sites can double sales. It can make a big difference for other products as well. Often help desk expenses can be dramatically reduced by an effort to deploy only software with high usability scores.

At Microsoft, an application feature is not considered complete until it has been usability tested, and the feature developers are expected to observe the test.[11] Not surprisingly, developers are often astonished at how users actually use the features they develop. There probably isn't any better way for developers to get rapid, unbiased feedback on how usable their approach is than to watch a usability test of a feature shortly after they write the code. In fact, the immediacy of the feedback is a primary factor in its effectiveness. A developer who has gone on to other things will not learn nearly as much from usability tests as one who observes users immediately after the feature is implemented.

Systems should be tested under load in the production environment or an exact replica, throughout the development cycle. One system I worked on would randomly overrun the thin client user buffers in production, something that could not occur in the development environment. In addition, a devious database transaction nesting problem led to system crashes, but only under the load of multiple users. These are the kinds of problems that are best discovered and corrected early in the development cycle, but they will only be discovered if the system is load tested in its target environment or a replica, early and often during development.

Principle 8: Avoid Sub-Optimization

If the principles of lean thinking seem counterintuitive, it is probably because our performance management systems tend toward management by disaggregation of a whole into its parts.[12] The problem is, the whole is

not the sum of its parts; in fact, optimizing individual parts has a tendency to sub-optimize the overall system. In our rush toward accountability and responsibility, we often create performance measurements that encourage optimization of parts at the expense of the whole. Yet once we have committed to measuring performance in pieces, it is often difficult to recognize the difficulties with those measurements.

When I was a systems manager in a magnetic tape manufacturing plant, performance was measured on unit cost and machine productivity. A huge part of unit cost was the burden rate of the coating machines. Assume a coating machine had to be depreciated at $100,000 per month. Then if the machine ran for 400 hours during the month, each hour had $250 burden rate. But if it ran for only 200 hours, then each hour had a $500 burden rate. Clearly, it was much better for the unit cost of all the products to run the machine for 400 hour each month.

However, if we ran the machine for 400 hours and produced product that was not immediately needed, we just built inventory that clogged the isles, got lost and damaged, and grew obsolete. We had no idea how much damage this inventory was doing to our operations until we started focusing on lowering inventory rather than reducing burden rate. We never realized how many hours we spent expediting orders that could not make it through the plant because of the piles of inventory, or the space we wasted storing inventory, or the time we spent keeping track of it. We had automated scheduling systems to help with the expediting and automated warehouses to handle all the stuff we pushed through those machines just to keep the burden rate low.

Eventually we implemented Just-in-Time scheduling. Much to our surprise, we found we needed half the production space for twice the production volume, and we could ship orders in a week rather than a month without expediting. We threw out the computer systems that scheduled, tracked, priced, and managed in-process inventory. The people who fed those systems data and looked after their health found better ways to spend their time. Amidst all of that efficiency, unit costs appeared to go up because the coaters were not running full time, the asset base of the

division was reduced because we had less inventory, and we had to write off the remaining depreciation on some computer systems. The accountants were not happy to see all of these numbers head in the wrong direction, yet the overall results of the division improved significantly.

Disaggregation

Conventional project management practice says we should disaggregate each project into a work breakdown structure (WBS), and manage the cost and schedule of the individual parts. This approach reminds me of the unit costs and burden rates in my plant, and how optimizing them buried us in hidden, unrecognized costs. The experience of lean manufacturing and concurrent product development should alert us to the fact that disaggregation is a seductive, but often sub-optimizing approach.

Managing a set of disaggregated tasks ignores the *flow of value across the entire economic chain*. It is concerned with the cost and time of *doing* things, but doesn't consider the value of *not doing* things. It looks within task boundaries, not at the impact of each task on the *end-to-end process*.

Usability is a good example of the importance of flow. It is easy to create individual screens for a web site, but until the *flow* of the user through the screens is considered, it is not clear if the individual screens, however artfully designed, are useful or a waste of time. Similarly, individual development steps such as testing or requirements gathering might be done very well, but if they are not integrated into the overall flow of development, they loose much of their usefulness.

The Project Scorecard

A decade ago, Robert S. Kaplan and David P. Norton[13] proposed the concept of *balanced scorecard* as a way to derive performance measures from the drivers of value in a company. Over the past ten years, balanced scorecards have been used in many large companies to "help guard against suboptimization.[14]"

The traditional project measurements of cost, schedule, and scope are financial measures, with the same tendency to create the same suboptimization that financial measures tend to create in other areas of the business. Corporate executives have adopted balanced scorecards because they understand that financial measures alone are inadequate and in fact, often focus attention on the wrong things. Instead of the assumption that all projects should be measured on cost, schedule, and scope, each project should develop an individual scorecard that reflects the drivers of business values for that project.

The first question to ask at the beginning of a project is "When this project is complete, who will decide if it is a success?" This should be followed closely by: "How will everyone know that this project is a success?" The answers to these questions should become the project scorecard. They should be the guidelines that influence the tradeoffs that people must make every day on the project.

Performance Measures

Measurements are funny things. Over time, you will get what you measure and pay attention to, so you have to be very careful to measure everything that is important. Thee trouble is, it's very difficult to measure everything, and when we notice that something is missing, we tend to add another measurement to plug the hole. A better approach to dealing with gaps in a measurement system is to *reduce* the number of measurements and *raise* the span of each measurement.

Even though our tendency is to disaggregate things to be sure we are measuring everything, we get better results with the opposite approach: measure the whole in preference to the pieces. Some people avoid aggregate measurements because they desire to hold individuals accountable, so they prefer to base performance measurements on things over which individuals have direct control. However, it is more effective to hold people accountable for things over which they have *influence*, not just what they can individually control. It is more effective to measure *team* performance, not *individual* performance.

It is rare that individuals can be successful by themselves, so rewarding them individually can create ill will among colleagues. On the other hand, if rewards are based on performance of a larger group, collaboration is encouraged. For example, a very large element of pay at Nucor Steel is based on productivity, but that productivity is measured across a large group of peers. Plant managers at Nucor, for example, receives a sizable amount of their salary based not on the productivity of their plants, but on the productivity of all the plants. Thus, they are encouraged to share their good ideas with all of the other plants.

Software development measures often involved defect counts, which give an indication of readiness for release. These measures should be aggregated and the entire team focused on reducing them. We learned long ago in manufacturing that defects are rarely the 'fault' of individuals; they are an indication of a problem with the end-to-end process. Thus everyone involved should work together to discover ways to reduce defects: developers, testers, analysts, even managers.

Across Company Boundaries

In 'Management Challenges for the 21st Century," Peter Drucker points out that the scope of management is not defined by the boundaries of an institution, "It has to be focused on results and performance across the entire economic chain."[15] He notes that in every case where a single management system has integrated the entire value chain, a cost advantage of 25 to 30% has resulted, resulting in dominance in the industry and the marketplace.[16]

For example, General Motors was created when William C. Durant bought up automotive companies and their suppliers, resulting in a vertically integrated company that made 70% of everything that went into a finished automobile. This vertical integration allowed GM to manage value across the economic value chain, and gave GM a 30% cost advantage over competitors for three decades.[17]

GM's began to loose this cost advantage about the same time that Japanese automakers developed the *keiretsu*, a tight-knit alliance of

affiliated companies that work toward each other's mutual success. The *keiretsu* resulted in the same cost advantage that GM had enjoyed, for the same reason; managing value across the economic chain gives a significant competitive advantage. It is neither vertical integration nor close supplier relationships that creates the competitive advantage; it is excellent management of value creation from the beginning to the end.

The 25 – 30% advantage Drucker attributes to vertical integration or a *keiretsu* comes from the removal of boundaries and the ability to create maximum benefit for the entire chain of organizations, rather the maximizing the individual benefit or each organization. Dell Computer is a good example of this. Recognizing that the greatest portion of value lies in the distribution end of the computer business, Dell controls that end of the business, while organizing its suppliers into a value chain that is significantly more efficient than its competitors.

The Purpose of Contracts

Unfortunately, the economic logic that drives close cooperation between companies in a supply chain is not widely recognized by people negotiating software development contracts. Most such contracts are aimed at protecting the parties from taking advantage of each other; they are generally silent on how to balance the overall good of the endeavor with the individual interests of each party. Such contracts do very little to help the parties manage end-to-end value creation across the economic chain. If we want to realize the significant economic benefits that accrue to those who focus on optimizing end-to-end value, we need to focus contracts on achieving the best results for the endeavor.

The first step to establishing a *keiretsu* is to reduce our dependence on using contracts as the vehicle that keeps parties from taking advantage of each other. Instead of using contracts for this purpose, we should depend upon the *relationship* to establish expectations and policies for fair behavior. Then we can focus the contract on creating the best overall value for the joint endeavor. *Good idea*, you say, *but how does it work in practice?*

Trust between firms does not come from trust between individuals; it comes from consistent, fair behavior over time. Thus, an effective partnership requires that *policies* requiring fair behavior be in place in each company. Companies that routinely and by policy focus on the overall good rather than their own individual advantage will behave in a consistent manner with their partners, even if individuals change. Consistent behavior creates confidence in partners and a reputation in the industry. This kind of confidence is not something that comes from contracts, but rather, it comes from a reputation built up by actions over time. If confidence in the fairness and capability of a partner is the basis of a partnership, then the contract can be focused on the overall good of the endeavor rather than keeping parties from taking advantage of each other.

Target Contracts

Target contracts provide a good mechanism for focusing on the overall good of the endeavor. A target contract establishes the overall targets that are important to success and provides in some way for an equitable sharing of the costs, benefits and risks of achieving the targets. The most common target contract is a target cost contract, in which both parties work to achieve the general goal of the contract within a specific cost. For example, Toyota will contract with tool and die makers to design and cut a stamping die for a total cost, including any changes. The details of the die are sketchy at the time of the contract, and the engineers in both companies will have to work closely together to achieve the target cost. If for any reason the target cost cannot be met, the companies will negotiate in good faith a fair way to share the overrun.

In software development, a target cost/schedule contract is often the most desirable contract form. In a target cost/schedule contract, both parties agree to try to achieve specific cost and schedule targets for achieving an overall objective. The tickly part about such contracts is that the details of the delivered features are not written into the contract; they are worked out through the close collaboration of developers with customers over the

course of the contract. If for any reason the targets cannot be met, the parties agree negotiate a fair resolution in good faith.

If target contracts seem risky, consider how risky the traditional software development contract has proven. Only a quarter to a third of projects succeed on all three fronts of cost, schedule, and scope at the same time.[18] Perhaps this abysmal record is not due to shortcomings in the projects themselves, but due to the assumption that it is possible and desirable to fix all three of these parameters and then predictably achieve them. If one can fix cost, schedule and objective (rather than scope) and obtain a high degree of certainty that the project will achieve success against these three targets, then target contracts can be considerably less risky than traditional contracts. If predictability is important, consider target cost/schedule contracts with competent partners that have a record of fairness.

Conclusion

Those who invest in financial instruments can obtain predictable results from an interest-bearing account. They receive low returns in exchange for letting someone else deal with the underlying uncertainty of financial markets. Those who want greater returns accept the fact that markets are uncertain and adopt strategies to obtain predictable results despite the unpredictable nature of the investment. The most common strategies are diversification and options.

If you are developing software for an evolving domain, you are better off accepting the underlying lack of certainty and adopting effective strategies to deal with it. Probably the least effective way to deal with uncertainty is to pretend that it isn't there.   This is the equivalent of investing all your money in a single stock and assuming it will continue to rise, or scheduling a wedding in August in Minnesota and assuming that it won't rain. No amount of planning can take away the inherent unpredictability of the weather or the performance of a single company.

Diversification is one of the most fundamental strategies for dealing with uncertainty in financial markets, and it works well for software development also. Avoid investments in monolithic systems, favor

strategies that develop systems one component at a time. Incremental development, particularly when coupled with incremental release to production, provides a way to break investments into small pieces that can be monitored separately.

Options are an even better way to deal with uncertainty. When you purchase an option you have the opportunity, but not the obligation, to do something in the future. If you rent a tent for that summer party, you are purchasing an option on a dry place to hold the party. You have the opportunity to set the tent up if it looks like rain, but you don't have to use it if the weather is beautiful.

Lean development is an options-based approach to software development. It focuses on starting early with an array of options and keeping those options open as long as possible so that final decisions can be made as late as possible. Options are explored through a series of end-to-end learning cycles that involve everyone who might have information to contribute. Once decisions are made, it is the mark of excellence to be able to implement them rapidly, consistently, and in a manner which yields the best system-wide results.

---

[1] In his Keynote 'ROI, It's Your Job,' at the Third International Conference on Extreme Programming, Alghero, Italy, May, 26-29, 2002, Jim Johnson, Chairman of The Standish Group, discussed the results of the annual Chaos Report by the Standish Group. It found that 28% of projects succeed, 23% fail outright, and 49% are 'challenged' – that is it completed over budget, over time, and/or with fewer features and functions than planned.

[2] Womack, James P., Jones, Daniel T., Roos, Daniel, *The Machine That Changed the World: The Story of Lean Production*, HarperPerennial, 1991; Originally published: Rawson Associates, New York, 1990. p 118.

[3] Dyer, Jeffrey H., *Collaborative Advantage, Winning Through Extended Enterprise Supplier Networks*, Oxford University Press; 2000 p 6

[4] Sobek, Durward K. II, Liker, Jeffrey K., Ward, Allen C., 'Another Look at How Toyota Integrates Product Development', *Harvard Business Review*, July-August 1988, p 44.

[5] This is the basic thesis of the book: Clark, Kim B, Fujimoto, Takahiro, *Product Development Performance; Strategy, Organization, and Management in the World Auto Industry*, Harvard Business School Press, Boston, 1991

[6] See footnote 5.

[7] See Collins, Jim, *Good to Great: Why Some Companies Make the Leap... and Others Don't*, Harper Business, 2001, Chapter 6.

[8] In his Keynote 'ROI, It's Your Job,' at the Third International Conference on Extreme Programming, Alghero, Italy, May, 26-29, 2002, Jim Johnson, Chairman of The Standish Group, discussed the results of a study of how often features and functions in a system are used: never (45%), rarely used (19%), sometimes used (16%), often used (13%), always used (7%).

[9] Quote from Sobek, Durward K. II, Liker, Jeffrey K., Ward, Allen C., 'Another Look at How Toyota Integrates Product Development', *Harvard Business Review*, July-August 1988, p 49

[10] Collins, Jim, *Good to Great: Why Some Companies Make the Leap... and Others Don't*, Harper Business, 2001

[11] Cusumano, Michael A., Selby, Richard W., *Microsoft Secrets, How the World's Most Powerful Software Company Creates Technology, Shapes Markets, and Manages People*, paperback edition, Simon & Schuster, 1998. Originally published in 1995

[12] Austin, Robert D., *Measuring and Managing Performance in Organizations,* 1996, Dorset Publishing House

[13] Kaplan, Robert S., Norton, David P., "The Balanced Scorecard – Measures That Drive Performance", *Harvard Business Review,* January- February 1992.

[14] Ibid

[15] Drucker, Peter F., *Management Challenges for the 21st Century,* Harper Business, 1999, p 34

[16] Ibid. p 33.

[17] Ibid

[18] See footnote 1.