Lean Software Development

C++ Magazine
Methodology Issue
(Publication Fall 2003)

Mary Poppendieck

I often hear developers say: *Life would be so much easier if customers would just stop changing their minds.* In fact, there is a very easy way to keep customers from changing their minds – give them what they ask for so fast that they don't have the *time* to change their minds.

When you order something on-line, it's usually shipped before you have time for second thoughts. Because it ships so fast, you can wait until you're sure of what you want before you order. But once you place the order, you have very little time to change your mind.

The idea behind lean thinking is exactly this: Let customers delay their decisions about exactly what they want as long as possible, and when they ask for something, give it to them so fast they don't have time to change their minds.

Sure, you say, *this is fine for Amazon.com, but how does it relate to software development?* The way to deliver things rapidly is to deliver them in small packages. The bigger the increment of functionality you try to deliver, the longer it takes to decide what is needed and then get it developed, tested, and deployed. Maintenance programmers have known this for years. When a piece of production code breaks, they find the cause, create a patch, test it rigorously, and release it to production – usually in the space of a few hours or days.

But, you say, *development is different; we need to develop a large system, we need a design, and we can't deploy the system piecemeal.* Even large systems that have to be deployed all-at-once should be developed in small increments. True, design is needed, but there is scant proof that great designs are achieved by meticulously gathering detailed requirements and analyzing them to death. Great designs come from great designers, and great designers understand that designs emerge as they develop a growing understanding of the problem.

As Harold Thimbleby said in *Delaying Commitment*,[1] "In many disciplines, the difference between amateurs and experts seems to be that experts know how to delay their commitments…. Amateurs, on the other hand, try to get things completely right the first time and often fail… in their anxious desire to avoid error, they make early commitments – often the wrong ones… In fact, the expert's strategy of postponing firm decisions, discovering constraints, and then filling in the details is a standard heuristic to solve problems."

Lean Thinking

Over the last two decades, Lean Thinking has created fundamental transformations in the way we organize manufacturing, logistics, and product development. For example, concurrent development has replaced sequential development in industries from airlines to automobiles, cutting product development time by perhaps a third and development cost in half, while improving the quality and timeliness of the product. There are those

[1] IEEE Software, May 1988

who think that rapid development is equivalent to shoddy work, but lean organizations have demonstrated quite the opposite. The measure of the maturity of an organization is the speed with which it can reliably and repeatedly respond to a customer request.

Yes, you heard that right. Maturity is not measured by the comprehensiveness of process documentation or the ability to make detailed plans and follow them. It is measured by operational excellence, and the key indicator of operational excellence is the speed with which the organization can reliably and repeatedly serve its customers. Thirty years ago, Frederick W. Smith, the founder of FedEx, envisioned an overnight delivery system the seemed ridiculous to most people; today even the post office offers overnight delivery. A decade ago, Toyota could develop a new car twice as fast as GM; today all automotive companies measure and brag about the speed with which they can bring a new car to market.

Let's revisit those customers you wish would make up their minds and stick to their decisions. Ask yourself this question: Once they do make up their minds, how fast can you reliably and repeatedly deliver what they want? Perhaps the problem does not lie in customers who can't make up their minds and keep changing their decisions. Perhaps the problem lies in asking them to decide well before they have the necessary information to make a decision, and in taking so long to deliver that their circumstances change.

Principles of Lean Software Development

There are seven principles of Lean Software Development, drawn from the seven principles of Lean Thinking. These principles are not cook-book recipes for software development, but guideposts for devising appropriate practices for your environment. [2] These lean principles have led to dramatic improvements in areas as diverse as military logistics, health care delivery, building construction, and product development. When wisely translated to your environment, they can change your basis of competition.

Eliminate Waste

All lean thinking starts with a re-examination of what waste is and an aggressive campaign to eliminate it. Quite simply, anything you do that does not add value from the customer perspective is waste. The seven wastes of software development are:

- → Partially Done Work (the "inventory" of a development process)
- → Extra Processes (easy to find in documentation-centric development)
- → Extra Features (develop only what customers want right now)
- → Task Switching (everyone should do one thing at a time)
- → Waiting (for instructions, for information)
- → Handoffs (tons of tacit knowledge gets lost)
- → Defects (at least defects that are not quickly caught by a test)

[2] The book *Lean Software Development: An Agile Toolkit* by Mary Poppendieck and Tom Poppendieck, Addison-Wesley Professional, 2003, provides twenty two tools for converting lean principles into agile software development practices.

All lean approaches focus on eliminating waste by looking at the *flow* of value from request to delivery. So if a customer wants something, what steps does that customer request have to go thorough to get delivered to the customer? How fast does that process *flow*? If a customer request waits in a queue for approval, a queue for design, a queue for development, a queue for testing, and a queue for deployment, work does not flow very fast. The idea is to create cells (or teams) of people chartered to take each request from cradle to grave, rapidly and without interruption. Then value *flows.*

Amplify Learning

The game of development is a learning game: hypothesize what might work, experiment to see if it works, learn from the results, do it again. People who design experiments know that greatest learning occurs when half of the experiments fail, because this exposes the boundary conditions. A development environment is *not* a place for slogans such as:

→ Plan The Work And Work The Plan
→ Do It Right The First Time
→ Eliminate Variability

The learning that comes from short feedback loops is critical to any process with inherent variation. The idea is not to eliminate variation, it is to adapt to variation through feedback. Your car's cruise control adapts to hilly terrain by frequently measuring the difference desired speed and actual speed and adjusting the accelerator. Similarly, software development uses frequent iterations to measures the difference between what the software can do and what the customer wants, and makes adjustments accordingly.

A lean development environment focuses on increasing feedback, and thus learning. The primary way to do this in software development is with *short*, *full-cycle* iterations. Short means a week to a month. Full cycle means the iteration results in working software: tested, integrated, deployable code. There should be a bias to deploy each iteration into production, but when that is not possible, end users should simulate use of the software in a production-equivalent environment.

We have known for a long time that iterative (evolutionary) development is the best approach for software development. In 1987 the report of the Defense Science Board Task Force on Military Software noted: "Document-driven, specify-then-build approach lies at the heart of so many DoD software problems…. Evolutionary development is best technically, and it saves time and money."[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. Economic markets develop options as a way to deal with uncertainty. Farmers, for example, can buy an option on the future price of

[3] Craig Larman, "A History of Iterative and Incremental Development", IEEE Computer, June 2003

grain. If the price drops, they have protected their profits. If the price raises, they can ignore the option and sell at the higher price. Options let people delay decisions until they have more information.

There are a lot of ways to keep options open in software development. Here are a few:

- → Share partially complete design information.
- → Organize for direct, worker-to-worker collaboration.
- → Develop a sense of when decisions must be made.
- → Develop a sense of how to absorb changes.
  - ✓ Avoid Repetition
  - ✓ Separate Concerns
  - ✓ Encapsulate Variation
  - ✓ Defer Implementation of Future Capabilities
- → Commit to Refactoring
- → Use Automated Test Suites

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. Similarly, when Just-in-Time concepts surfaced in Japan in the early 1980's, most western manufacturers could not fathom why they made sense. Everybody *knew* that the way to make customers happy was to have plenty of product on the shelf, and the way to maximize profits was to build massive machines and keep them busy around the clock. It took a long time for people to realize that this conventional wisdom was wrong.

The goal is to let your customer take an options-based approach to making decisions, letting them delay their decisions a long as possible so they can make decisions based on the best possible information. Once your customers decide what they want, your goal should be to create that value just as fast as possible. This means no delay in deciding what requests to approve, no delay in staffing, immediate clarification of requirements, no time-consuming handoffs, no delay in testing, no delay for integration, no delay in deployment. In a mature software development organization, all of this happens in one smooth, rapid flow in response to a customer need.

Empower the Team

In a lean organization, things moves fast, so decisions about what to do have to be made by the people doing the work. Flow in a lean organization is based on local signaling and commitment amongst team members, not on management directives. The work team designs its own processes, makes its own commitments, gathers the information needed to reach its goals, and polices itself to meet its milestones.

Wait a minute, you say. *Why would I want to be empowered? If I make the decisions, then I'll get blamed when things go wrong.* A team is not empowered unless it has the training, expertise, and leadership necessary to do the job at hand. But once those

elements are in place, a working team is far better equipped to make decisions than those who are not involved in the day-to-day activities of developing software. It is true that decision-making carries greater responsibility, but it also brings significantly greater influence on the course of events and the success of the development effort.

Consider an emergency response unit such as firefighters. Members receive extensive training, both in the classroom and on the job. Exercises and experience imprint patterns of how to respond to difficult situations. When an emergency occurs, the team responds to the situation as it unfolds; there is little time to ask remote commanders what to do, nor would it make sense. The important thing is that the responding teams have the training, organization and support to assess the situation as they encounter it and make basically correct decisions on their own.

Similarly in software, the development team is in the best position to know how to respond to difficult problems and urgent requests. The best way to be sure that you get things right is to work directly with customers to understand their needs, collaborate with colleagues to figure out how to meet those needs, and frequently present the results to customers to be sure you are on the right track. Management's job to supply the organization, training, expertise, leadership, and information so that you generally make the right decisions, make rapid course corrections as you learn, and end up with a successful outcome.

Build Integrity In

There are two kinds of integrity – perceived integrity and conceptual integrity. Software with perceived integrity delights the customer – it's exactly what they want even though they didn't know how to ask for it. Google comes to mind – when I use Google I imagine that the designers must have gotten inside my head when they added the spelling checker. I would never have known to ask for this feature, but these days I type most URLs into the Google toolbar because if I mistype, Google will set me straight.

The way to achieve perceived integrity is to have continuous, detailed information flow from the users, or user proxies, to the developers. This is often done by an architect or master designer who understands the user domain in detail and makes sure that the developers always have the real user needs in front of them as they make day-to-day design decisions. Note that the technical leader facilitates information flow and domain understanding, she or he is intimately involved in the day-to-day work of the developers, keeping the customer needs always in mind. However, it is the developers, not the leader, who make the detailed, day-to-day decisions and tradeoffs that shape the system.

Conceptual integrity means that all of the parts of a software system work together to achieve a smooth, well functioning whole. Software with conceptual integrity presents the user with a single metaphor of how a task is to be done. You don't have one way of buying airline tickets if you are paying with cash and a totally different way if you are using frequent flier miles. You don't use kilometers in one module and miles in another.

Conceptual integrity is achieved through continuous, detailed information flow between various technical people working on a system. There are no two ways about it, people have to *talk* to each other, early and often. There can be no throwing things over the wall, no lines between supplier, development team, support team, customer. Everyone should be involved in detailed discussions of the design as it develops, from the earliest days of the program.

For example, Boeing credits its rapid and successful development of the 777 to its 'Working Together' program, where customers, designers, suppliers, manufacturers, and support teams all met from the start to design the plane. Early on it was discovered that the fuel tank was beyond the reach of all existing refueling trucks, a mistake that was easily fixed. In a normal development process this expensive mistake would not have been discovered until someone tried to fuel the first plane.

There are those who believe that software integrity comes from a documentation-centric approach to development: define the requirements in detail and trace every bit of code back to those requirements. In fact, such an approach tends to interfere with the technical communication that is essential to integrity. Instead of a documentation-centric approach, use a test-centric approach to integrity. Test early, test often, test exhaustively, and make sure an automated test suite is delivered as part of the product.

See the Whole

When you look at them closely, most theories of how to manage software projects are based on a theory of disaggregation: break the whole into individual parts and optimize each one. Lean thinking suggests that optimizing individual parts almost always leads to sub-optimized overall system.

Optimizing the use of testing resources, for example, decreases the ability of the overall system to rapidly produce tested, working code. Measuring an individual's ability to produce code without defects ignores the well-known fact that about 80% of defects are caused by *the way the system works*, and hence are management problems.

The best way to avoid sub-optimization and encourage collaboration is to make people accountable for what they can *influence*, not just what they can *control*. This means measuring performance *one level higher* than you would expect. Measure the team's defect count, not that of individuals. Make testing as accountable for defect-free code as developers. To some it seems unfair to hold a team accountable for individual performance, but lean organizations have found that individuals are rarely able to change the system which influences their performance. However a team, working together and responsible for its own processes, can and will make dramatic improvements.

Conclusion

To keep customers from changing their minds, raise the maturity of your organization to the level where it can reliably deliver what customers want so fast that they have no time to change their minds. Focus on value, flow, and people, the rest will take care of itself.

