What Engineering Has in Common With Manufacturing and Why It Matters

Dr. Alistair Cockburn
Humans and Technology

Software engineering is more like manufacturing than most people expect. Once we spot their similarities, we can apply the lessons learned over the last 50 years in manufacturing to software development. This article picks six lessons to apply to software development gleaned from the manufacturing industry.

It is generally considered frivolous to compare engineering – software engineering, in our case – with manufacturing. Manufacturing (so the reasoning goes) consists of making the same thing over and over, while software engineering is about making something different each time. In software engineering, coming up with the design and code is the hard part, while production is the easy part, sometimes as easy as publishing to the internet.

Software engineering is remarkably similar to manufacturing once we notice *decisions* as the product that moves through a network of people. In software development, people make decisions, hand those decisions to other people to build on, and (most importantly for this article) wait for other people to make their decisions. The *decision* in software development corresponds to a *part* in a manufacturing line: Both flow through a network, wait in queues at bottlenecks, have throughput delays, and so on.

With this equivalence in place, there is a very real parallel between design and manufacturing. This is useful to us because manufacturing has been studied heavily over the last 100 years, and we can learn from lessons in that industry.

In what follows, I shall focus on software development, but it should be clear that the same argument applies to every team-design activity, including engineering, theatre, publishing, and much of business.

Waiting for Decisions

We start by recognizing that in team-design activities, people wait on each other for decisions.

Figure 1 shows a simplified view of the dependencies between people in software development (it is missing the feedback loops, in particular). Figure 2 shows a more complete mapping of the decision dependencies, with some typical feedback loops. The feedback loops complicate matters, but do not change the basic results.

In Figure 1, the dependency of one person on another is shown with a large black arrow. The person at the tail of the arrow is making decisions and passing them to the person at the head of the arrow. A small pyramid represents the actual decision being passed from one person to another.

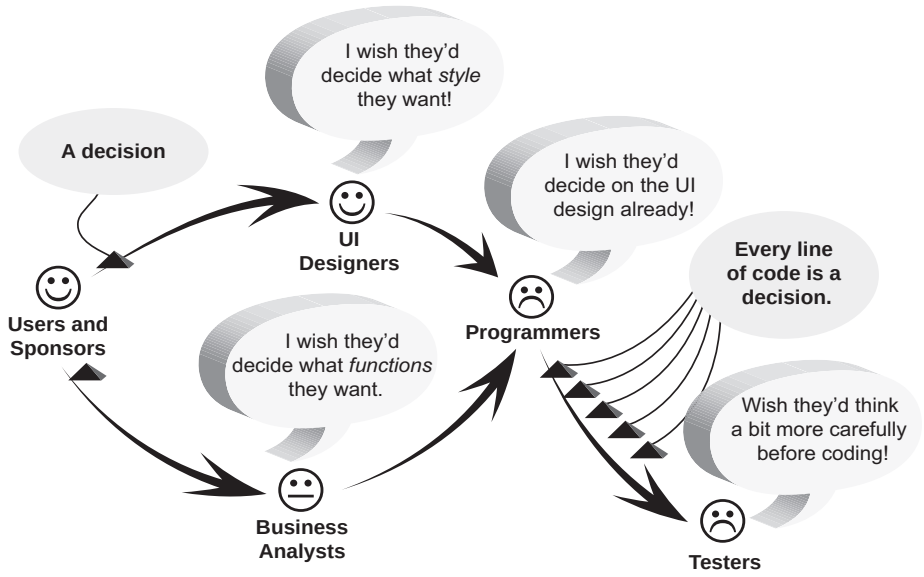
In Figure 1, we see the following:

- Business analysts and user interface (UI) designers waiting for users and sponsors to decide what functions and design styles they want.
- Programmers waiting for business analysts to work out the business rules and UI designers to allocate behavior to different pieces of the user interface.
- Testers waiting for programmers to finish their coding.

A nice thing about considering individual *decisions* as connecting people is that we can move away from stereotypes about how a company's process or decision-making activities *ought* to look, and instead focus on how it *actually* looks – what decisions actually get made by which people, and who is really waiting for whom.

There is no ideal software process any more than there is any ideal manufacturing process. Each company has its own strong-minded people who make a disproportionate number of decisions that might, in other companies, be made by people in other roles. Each company has its own shortage of UI designers, programmers, testers, or even sponsors, which causes its process to have a certain characteristic shape – people working overtime or sitting idle because other people can not get their work done fast enough. Each company has its own reasons to have a large, external test department, or perhaps no test department at all.

Figure 1: *People Wait on Other People for Decisions*



Different Bottlenecks, Different Processes

In any organization, we can find a backlog of decisions stacking up at some particular work group. This creates a *bottleneck*, which limits the speed of the overall team. Bottlenecks are of great concern in manufacturing and have received much study. The obvious thing to do is to increase the capacity of the bottleneck group – hire more people, or better people, or get better tools, and so on.

Sooner or later, however, the organization hits its limit as to what it can do to improve the speed of the bottleneck group. At that point, what comes into play is the process definition itself.

Figure 3 shows three different, but fairly typical organizations.

In the first organization, there are not enough UI and database designers to keep up with the work. We see decisions stacked up at their work centers. Assuming that this organization cannot or will not hire more UI and database designers, it should look at ways to have programmers and business analysts pick up sections of the UI designer's and database designer's work. Even assuming that UI design work is specialized, parts of that work can be automated, carried out by assistants, or handled by programmers.

In the second organization, there are not enough experienced programmers, and work requests stack up in front of them. In this second organization, the reverse is more the case. The programmers, being few and inexperienced, might need to have much of the problem *digested* for them as often as possible.

In such a situation, in which I participated, we recommended that the business analysts write quite detailed use cases (not containing the user interface, but containing the business rules more explicitly than we otherwise might), the layouts of the data needs, plus discussions of different business scenarios. The business analysts sat with the respective programmers as they started on each use case and discussed the use case, the scenarios, and the data. The business analysts left the paperwork with the programmers and made themselves available for discussions and tutorials as needed.

The process we came up with was aimed at minimizing the trouble the programmers had to undergo to understand the problem at hand. This is quite different from the process in the first organization.

In the third organization, the users and sponsors are notably missing from the discussion. What happens in these organizations is that the business analysts and UI designers end up making the business decisions and then sending those decisions (or running products) back to the users and sponsors for comment. The picture shows those requests for review stacking up in front of the users and sponsors.

The third picture also shows the programmers and database designers sending decisions back and forth to each other. Both groups need to come to agreement on the domain model and how that will be represented in the code and in the database.

In the third organization, the process might call for prototypes and early samples to be produced and put in front of the users and sponsors. Since those people have the least availability, the material should be as fully prepared as possible. Also, since close collaboration between the programmers and database designers

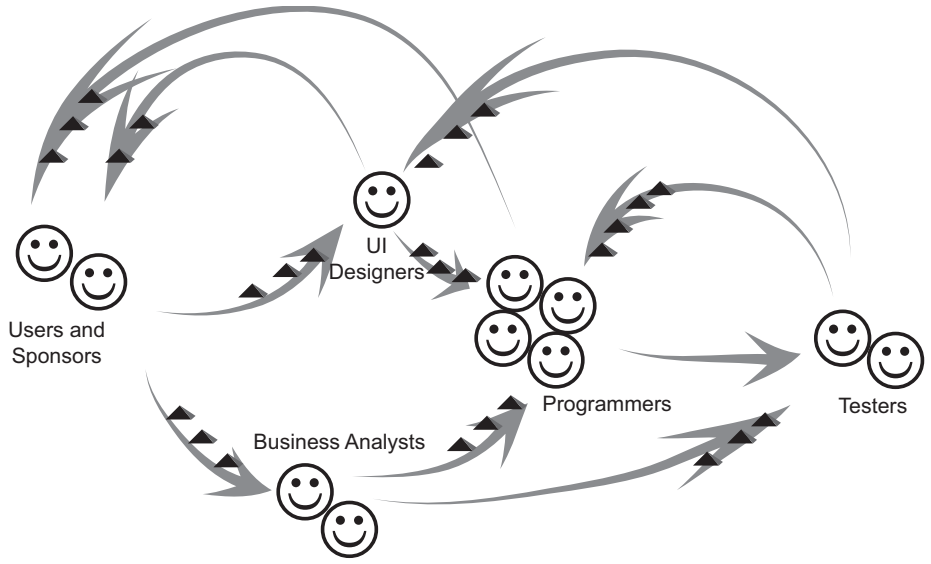


Figure 2: *Optimal Process and Strategies Vary With the Decision-Dependency Network*

is required, those teams should be seated together, or at least have frequent meetings and joint design reviews.

There are the following two points to draw from these pictures:

- The organizations should be using different processes.
- These drawings help us to see how those processes should be different.

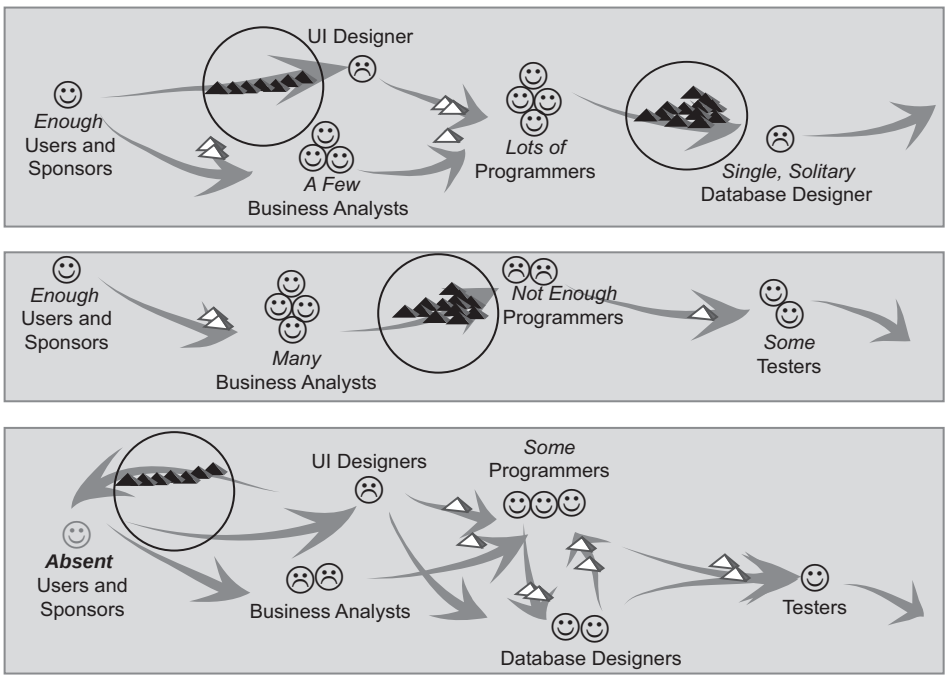
Lessons From Manufacturing

To apply the lessons from manufacturing, we need to recognize the life cycle of a decision:

- The decision gets made. It might be a business-level decision, a UI-design decision, or a decision about a particular line of code. The person making the decision does not really know at this point if it is a good decision or not.
- The decision gets reviewed internally. Part of reviewing a line of code is passing it through a test suite. Part of reviewing a UI design is putting it in front of a group of test users. Part of reviewing a business decision is putting it in front of sponsors and test markets. The decision fails the review, gets marked for adjustment, or passes.
- The decision gets pushed out into the world. At this point, the world makes a judgement about the quality of the decision and the decision makers get very useful feedback.

Even a very good decision has a finite lifetime, after which time it needs adjusting. A major goal of the development team is to get decisions reviewed, repaired, and sent

Figure 3: *A More Complete View of a Decision Dependency Network*



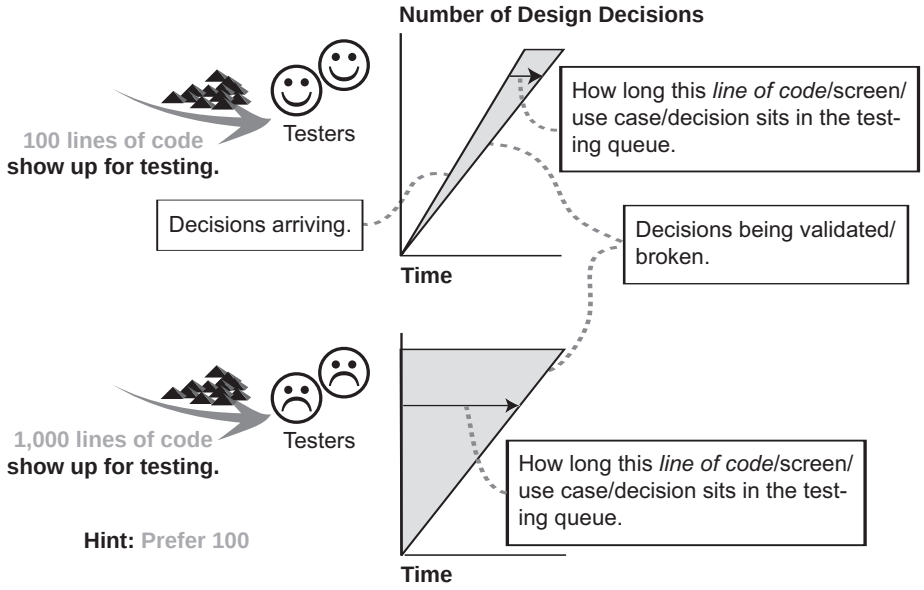


Figure 4: *Feed Systems Run More Smoothly With Small Transfer Sizes*

out into the world earning value as soon as possible. All the decisions waiting for internal and external review constitute internal inventory or work in progress (WIP).

Move Inventory Out

One of the lessons to draw from manufacturing is to reduce the WIP, that is, *get decisions out of development and into the business*. This is important in manufacturing, and it is also important in software development, because the value of decisions decays over time. Every moment a decision stays in the development cycle costs the organization money.

- Each *requirement* is a decision based on a business climate. When the business climate changes, the decision may become incorrect. If the software is not yet earning value for the company, the requirement is a waste.
- An *architecture* is a decision based on technology and business. If the technology changes before the software is earning value for the company, those decisions are a waste.
- Each line of code is a decision based on requirements, domain, technology, and aesthetics. If anything causes it to become obsolete before the software is earning value for the company, it is a waste.

To the extent that it is not earning value in the business, each decision loses value and quality with time. The more decisions stuck inside the pipeline, the more *decaying inventory* the organization is carrying.

Inventory stacks up quickly. Assume for reference an organization that is so fast that when a new requirement arrives, it can implement and deploy it by the next morning:

- The company with a one-day turnaround has about one day's worth of inventory lying around the office.
- The company with a two-week turnaround has about 10 days worth of inventory lying around the office.
- The company with a quarterly delivery system (assuming they deploy from fresh requirements every quarter) has about 100 days of inventory lying around.
- The company delivering a three-year project has 1,000 days of inventory (decaying) around them.

The message, in software as much as in manufacturing, is the following: *Get the inventory out the door and earning value!* Find ways to shorten and speed the pipeline.

Move Small Amounts, Continuously

The next lesson to draw from manufacturing is that, for the WIP (decisions still inside development), reduce the size of transfers between groups. Move small amounts often rather than stacking them up in large batches for long periods of time.

Figure 4 shows two ways of transferring work from the programmers to the testers.

In the first case, the programmers hand over 100 lines of code (each week, let's suppose). The testers get a regular weekly arrival rate of about 100 lines of code and have to integrate and test them against the rest of the system and against the known defect log.

The amount actually handed over will vary, of course, and the actual length of time needed to work through the new code will also vary. That variance is part of why small amounts should be handed over at any one time.

The lower part of Figure 4 shows the programmers handing about 1,000 lines of code to the testers (each quarter, that would be, to keep the rate of production about the same as in the upper picture).

The problem with the lower picture is that all 1,000 lines of code show up at one time. The responsiveness of the testing group suddenly becomes much more variable with the large arrival of an unknown number of bugs of varying sizes.

Equally bad is when the testers start handing bug reports back to the programmers and the programmers suddenly see a large spike of requests on their input queue coming from the testers (see the arrow in Figure 2, from the testers back to the programmers). The programmers are now juggling two work queues: requests for new features and requests for bug fixes.

Manufacturing groups have experienced and studied all the previous examples, and they concluded that these sorts of feed systems run best when *small* amounts of work get handed from one group to the next. The ultimate goal is to hand over just one part from one group or person to the next.

Toyota pioneered this idea in its lean or just-in-time manufacturing lines. They aim for *continuous flow*, the flow of just one piece of material from one person to another (what to do when a queue backs up is the subject of another lesson).

It is not clear exactly what *continuous flow* might mean in software development. Some design decisions affect large parts of the system, and some decisions can not be validated for a long time. However, the experiences in manufacturing are backed up by both mathematical models and experiences in agile software development.

It is rare to find development teams able to deploy fresh requirements every week, but I have been able to find a few teams who both deploy weekly *and* have a low enough defect rate that they get only a few requests a day. On one team I talked with, a person was assigned each day, on a rotation, to handle any incoming requests, whether bug reports or requests for small enhancements. That person would stop other work, do the work and redeploy the system before rejoining the main group. The average time to re-deployment was half a day. With such a small, steady flow of requests on the feedback queue, the team was able to keep from being diverted from their main assignment.

Cross-Train People

The literature on lean and agile manufacturing contains the recommendation to cross-train (training in multiple areas) people at adjacent stations. The idea is that when a small bubble of inventory shows

up at someone's input, the neighboring person, having a spare moment, steps over and works it down. In this manner, small variances in work flow can be evened out and not disturb the organization's overall flow.

We see this in software development when programmers help testers, user interface designers, and business analysts, or when business analysts help testers. Unfortunately, programming is a technical enough activity that UI designers, testers, and business analysts are unlikely to be able to help the programmers. Programmers can help other programmers, though. We see in some companies that front-end developers, middle-ware developers, and back-end developers help each other when one of the groups has a sudden bump in work.

Extend the Network

All of these ideas are good – so good, in fact, that people using them soon find that their bottleneck lies somewhere in their supply chain, whether sponsors, subcontractors, or distributors. They start to draw the dependency network for the larger system in which they sit, and they start including their supply chain partners in their discussions.

Toyota is well known for working with its suppliers. Less well known are cases of software development groups doing it. The same team I referred to earlier, using the daily programmer rotation for fixes and enhancements, also wrote automated acceptance tests for their subcontractor's part of the system. They reasoned that their time was better spent writing automated acceptance tests and catching bugs on arrival than debugging and finding those same faults in the integrated system when their supplier's code broke. The supplier was, of course, surprised and delighted to find they did not have to write the automated acceptance tests.

The lesson from Toyota and the other companies who are streamlining the wider network is the following: *The wider the network of* we*, the faster we all go.*

Who's Writing About This?

Once we see the mapping between manufacturing and team design activities, suddenly a lot of literature becomes available.

Toyota's production system (also called The Toyota Way and lean manufacturing) is widely documented. A good place to start is with *The Toyota Way Fieldbook* [1].

The application of lean manufacturing principles to design work is described in *Managing the Design Factory* [2], *Product Development for the Lean Enterprise* [3], and *The Elegant Solution* [4].

Tom and Mary Poppendieck describe in several books [5, 6] how lean manufacturing principles fit software development. *Agile Software Development: The Cooperative Game* [7] contains an experience report from a software product company (Tomax) that includes its customers in its dependency network.

Elihu Goldratt wrote about bottleneck stations in manufacturing [8] and then widened the discussion to constraints in general (*theory of constraints*) [9]. David Anderson applied the theory of constraints and queue size to software projects [10]. I have written about strategies for dealing with bottlenecks that have reached their capacities [11].

Summary

It is not immediately obvious that software development teams can learn from manufacturing. However, once we chart the network of dependencies between people in a software development organization and make the shift to think of *decisions* as comprising the team's *inventory*, then the parallels become startlingly clear.

We learn six lessons from the parallels:

- Drawing the decision-dependency network helps us spot the *bottleneck* stations, where decisions-to-be-made are piling up.
- From the different decision-dependency networks in various organizations, and their varying bottlenecks, we can see how the *optimal process* varies from organization to organization.
- *Move inventory out.* Decisions decay over time, so it is important to find ways to shorten the pipeline from arrival of a request or decision to the deployment of the system.
- *Move small amounts, continuously.* Transferring large amounts of inventory (decisions, in our case) between workers causes unpredictable variations in the organization's output. It is better to move small numbers of decisions more often. This reinforces the idea of incremental development, with the smallest increment size possible.
- *Cross-train people.* When people can help each other across specialties, they can move quickly to eliminate small bubbles in each others' input queue, thus smoothing the organization's output.
- *Extend the network.* By widening the network included in the dependency analysis and queue-size reduction, a company can smooth its own input stream and simplify its work.◆

References

1. Liker, J., and D. Meier. The Toyota Way Fieldbook. McGraw-Hill, 2005.
2. Reinertsen, D. Managing the Design Factory. Free Press, 1997.
3. Kennedy, M. Product Development for the Lean Enterprise. Oaklea Press, 2003.
4. May, M. The Elegant Solution. Free Press, 2006.
5. Poppendieck, M., and T. Poppendieck. Implementing Lean Software Development: From Concept to Cash. Addison-Wesley, 2006.
6. Poppendieck, M. and T. Poppendieck. "Lean Software Development." C++ Magazine (Fall 2003).
7. Cockburn, A., Agile Software Development: The Cooperative Game. 2nd Edition. Addison-Wesley, 2006.
8. Goldratt, E. The Goal. North River Press, 1992.
9. Goldratt, E. Theory of Constraints. North River Press, 1999.
10. Anderson, D. "Managing Lean Software Development With Cumulative Flow Diagrams." Proc. of the Borland Conference, 11-15 Sept. 2004, San Jose, CA.
11. Cockburn, A., "Two Case Studies Motivating Efficiency as a Spendable Quantity." Humans and Technology Technical Report HaT TR 2005.00 <http://alistair.cockburn.us/index.php/Two_Case_Studies_Motivating_Efficiency_as_a_%22Spendable%22_Quantity>.

About the Author

Alistair Cockburn, Ph.D., is an expert on object-oriented (OO) design, software development methodologies, use cases, and project management. He is the author of *Agile Software Development*, *Writing Effective Use Cases*, and *Surviving OO Projects* and was one of the authors of the Agile Development Manifesto. Cockburn defined an early agile methodology for the IBM Consulting Group, served as special advisor to the Central Bank of Norway, and has worked for companies in several countries. Many of his materials are available online at <http://alistair.cockburn.us>.

1814 East Fort Douglas CIR
Salt Lake City, UT 84103
Phone: (801) 582-3162
Fax: (775) 416-6457
E-mail: acockburn@aol.com