# The Agile Enterprise and the Division of Labor

Gene Callahan

NYU Tandon School of Engineering

Leonard E. Read began his famous essay “I, Pencil” (<http://www.econlib.org/library/Essays/rdPncl1.html>) by noting:

I, Pencil, simple though I appear to be, merit your wonder and awe, a claim I shall attempt to prove… Simple? Yet, not a single person on the face of this earth knows how to make me. This sounds fantastic, doesn't it? Especially when it is realized that there are about one and one-half billion of my kind produced in the U.S.A. each year.

Read goes on to list just a few of the many, many people who contribute to the making of a “simple” pencil: loggers, miners, makers of chain saws, hemp growers, the manufacturers of railroads and railroad cars, millworkers, producers of precision assembly-line machines, the harvesters of canola seed, farmers growing castor beans, and more.

What Read is praising in his essay are the benefits of the division of labor, the economic process through which a human community, by dividing up tasks and “assigning” various members to specialize in each task, can greatly increase its output. (I put “assigning” in scare quotes because, in a market economy, for the most part people are not literally assigned to tasks, but instead choose their roles in the division of labor based upon their talents and the prevailing compensation for each possible role they could fill.) The benefits of the division of labor were, of course, recognized at least as far back as Plato and Xenophon. As Plato put it in *The Republic*, “Well then, how will our state supply these (physical) needs? It will need a farmer, a builder, and a weaver, and also, I think, a shoemaker and one or two others to provide for our bodily needs. So that the minimum state would consist of four or five men.” And Adam Smith famously expounded upon those benefits in *The Wealth of Nations*, writing “The greatest improvement in the productive powers of labour, and the greater part of the skill, dexterity, and judgment with which it is anywhere directed, or applied, seem to have been the effects of the division of labour” (http://www.econlib.org/library/Smith/smWN1.html#B.I,%20Ch.1,%20Of%20the%20Division%20of%20Labor).

In the early 20th century, this method of increasing productivity was pushed to its limits. Tasks were broken down to the extent that workers with minimal skills could be assigned simple, highly repetitive actions, and perform them with almost no knowledge of what anyone else on the assembly line was up to.

Although this led to higher output of standardized products, the disadvantages of extending the division of labor to this extent were not overlooked. Karl Marx noted that the extensive division of labor alienated the worker from the product he was producing: someone who spends all day tightening a particular lug nut may be little able to associate what they do with “making a car.” But even Adam Smith, typically understood as a great proponent of the division of labor, commented:

In the progress of the division of labour, the employment of the far greater part of those who live by labour, that is, of the great body of people, comes to be confined to a few very simple operations, frequently to one or two. But the understandings of the greater part of men are necessarily formed by their ordinary employments. The man whose whole life is spent in performing a few simple operations, of which the effects are perhaps always the same, or very nearly the same, has no occasion to exert his understanding or to exercise his invention in finding out expedients for removing difficulties which never occur. He naturally loses, therefore, the habit of such exertion, and generally becomes as stupid and ignorant as it is possible to become for a human creature to become.

Smith is pointing out a *general* problem with the extensive division of labor, but there is a much more *particular* problem, which only came to prominence in the recent days of increasing automation and increasing demand for innovative and customized products: the sort of mindless, production-line division of tasks common in mid-20th-century factories created a workforce downright *discouraged* from thinking about how their work fit into the production process as a whole, or how alterations in other parts they did not directly make might affect their own task. Such a holistic view was only supposed to be required of the engineers who designed new products or who designed the factory processes that would produce those new products. As in a socialist economy, all knowledge about the product and the production process would be concentrated at the top of a pyramid of work, and those below the peak were to just mindlessly follow the orders of those knowledge commissars.

A problem with this approach is that as products become more complicated and the pace of innovation increases, no single mind, or even a small group of minds, is capable of grasping all of the interconnections between the different parts of those complex products, and thus, cannot foresee how an innovation supposedly concerning only one part will actually have ripple effects on many other apparently separate production tasks. This fact was realized quite early at Toyota, and led to the invention of the [Toyota Production System](https://en.wikipedia.org/wiki/Toyota_Production_System), the forerunner of Lean Software Development. As Mary and Tom Poppendieck note in *Implementing Lean Software Development*:

Toyota’s real innovation is its ability to harness the intellect of “ordinary” employees. Successful lean initiatives must be based first and foremost on a deep respect for every person in the company, especially the “ordinary” people who make the product or pound out the code. (pp. 124-125)

As important as these ideas were in factory production, their importance is even greater in the world of software development, where production is *always* production of a novel product: otherwise, one would simply buy or rent an existing software product, which is almost always a lower cost venture than “rolling your own.”

In such an environment, it is simply not possible to assign the “workers” (programmers) a simple, repetitive task, and expect them to achieve decent results without at least some understanding of the overall product design, as well as an understanding of how their particular “part” integrates with the other parts of the product as a whole. In such a situation, worker obedience no longer “works.” A manager cannot tell a software engineer working on a product of even moderate complexity to just follow the manager’s orders: the programmer can bring production to a halt simply by asking, “OK, what line of code should I write next?”

But further: no knowledge worker producing an even moderately complex product can do his work properly without his understanding of his part in the production process evolving in continuous interaction with the evolving understanding of all of the other knowledge workers involved in the product: one such worker gaining a better understanding of the nature of her component simply *must* convey that understanding to all other workers upon whom the changes in her component have an impact, and that set of workers typically encompasses *everyone* working on the product. As the Disciplined Agile Framework has it:

Enterprise awareness is one of the key principles behind the Disciplined Agile (DA) framework. The observation is that DA teams work within your organization’s enterprise ecosystem, as do all other teams. There are often existing systems currently in production and minimally your solution shouldn’t impact them. Better yet your solution will hopefully leverage existing functionality and data available in production. You will often have other teams working in parallel to your team, and you may wish to take advantage of a portion of what they’re doing and vice versa. Your organization may be working towards business or technical visions which your team should contribute to. A governance strategy exists which hopefully enhances what your team is doing. (<http://www.disciplinedagiledelivery.com/enterpriseawareness/)>

The various aspects of Agile / Lean / DevOps production follow from the nature of knowledge workers cooperating to create innovative products. Programmers cannot do their jobs in isolation: thus, the practice of *continuous integration*, which quickly exposes mutual misunderstandings of how one person’s work impacts that of others. Testers cannot test successfully, without introducing large delays in deployment, unless they are part of the production process from day one: thus, *continuous testing*, guaranteeing that product flaws are exposed and fixed at the earliest moment possible. Operations cannot successfully deploy constantly evolving products unless deployment itself becomes a software product capable of evolving as fast as the products of the developers: thus, *software as infrastructure*. The “business” stakeholders in the product cannot ensure the product is really meeting business needs unless they are continually engaged in the development process: thus *continual interaction* between the engineers and the “business people.” How new versions of a piece of software impact the end users cannot be determined without continual feedback from those users: thus, *incremental development*, which means developer work on small batches and can easily change course; *continuous deployment*, allowing end users to comment on the work done in those small batches, and *continuous monitoring*, so that any problems using the product become known almost as soon as they occur.

A rigid division of labor hinders businesses from responding agilely to changing market conditions while producing software. If workers are confined to a narrow silos based on job title, the interaction between the many components of a complex piece of software must be defined from the top down, and this restriction will result in a very limited capacity to deviate from an initially defined pattern of interaction. In Disciplined Agile, it is noted:

IT departments are complex adaptive organizations. What we mean by that is that the actions of one team will affect the actions of another team, and so on and so on. For example, the way that your agile delivery team works will have an effect on, and be affected by, any other team that you interact with. If you’re working with your operations teams, perhaps as part of your overall DevOps strategy, then each of those teams will need to adapt the way they work to collaborate effectively with one another. Each team will hopefully learn from the other and improve the way that they work. (<http://www.disciplinedagiledelivery.com/agility-at-scale/disciplined-agile-2/>)

Let us consider a realistic change that might hit a project mid-stream, and just a few of the areas it might impact.

I was once developing an option-trading package for a team of traders. At first, we were only getting quotes for options from a single exchange. The traders realized that they wanted instead to see the best bid and ask from every exchange, which meant we needed to get quotes from four exchanges, not one. This might seem to be a specification change with a narrow scope: just add three more price feeds to the application. Who would this concern beyond the programmer who would be adding the feature?

Well, for one, it would concern the team supporting the price server: this was going to quadruple the load this application would place on it. It was also going to impact the order server: that server had to be prepared to send orders out to the proper exchanges. Oh, and the testing team had better be prepared to simulate quotes coming in from four sources, not one. Also, the monitoring team would have to detect if there was a lag on quotes arriving from four sources, not one.

Or consider the patterns and tales from Michael T. Nygard’s book, *Release It!*. Continually, in Nygard’s stories, solving a problem in a sophisticated web operation involves a wide range of both technical and business knowledge. For instance, in terms of designing “circuit breakers” that limit the impact of the failure of one component, Nygard notes that deciding what to do when a circuit breaker trips is not merely a technical decision, but involves a deep understanding of business processes: “Should a retail system accept an order if it can’t confirm availability of the customer’s items? What about if it can’t verify the customer’s credit card or shipping address?” (p. 97) Later in the book, a retail system went down entirely on Black Friday, costing his client about a million dollars an hour in sales. Fixing the problem involved understanding the functioning of the frontend of the online store, the order management system, and the scheduling system, and the interactions of the three.

A software engineer who thinks of his job narrowly, as just being responsible for writing the code to do the task he is told the code should do, is not going to be thinking of the multiple other areas this change would affect. And a higher-level designer is unlikely to know enough of the details of all of these areas to fully understand the impact of this change: the best bet for being able to successfully respond to this changed business requirement is for the people working in each specialization also to have a vision of the overall system, an understanding of how other specialized areas function, and to have robust communication channels open between the various specialties: in other words, to break down the silo walls produced by a rigid division of labor, and embrace agile development principles. Or, as said in Disciplined Agile:

However, to succeed delivery teams must often work with people outside of the team, such as enterprise architects, operations engineers, governance people, data management people, and many others. For agile/lean delivery teams to be effective these people must also work in an agile/lean manner. (<http://www.disciplinedagiledelivery.com/agility-at-scale/disciplined-agile-2/>)