PROJECT MYOPIA

A companion volume to Continuous Digital

Why Projects Damage Software

#NoProjects

ALLAN KELLY

AKA
Allan Kelly Associates

Project Myopia

Allan Kelly

This book is for sale at http://leanpub.com/myopia

This version was published on 2020-02-25

ISBN 978-1-912832-03-3

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

Tweet This Book!

Please help Allan Kelly by spreading the word about this book on Twitter!

The suggested tweet for this book is:

I just bought ”Project Myopia” a #NoProjects book by @allankellynet

The suggested hashtag for this book is #NoProjects.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

#NoProjects

Contents

Free Book

Xanpan is available for free

Xanpan: Team Centric Agile Software Development is available for free to all subscribers to Allan Kelly's newsletter[1]

[1] https://www.allankellyassociates.co.uk/xanpan_offer/

Project Myopia

> *Project Myopia*: The belief that the project model is the only way of managing business change and development. Not seeing digital development as a continuing commitment to growing the business, but instead believing it will end and working towards that end.

Type 1 Project Myopia: Success

Failing to recognize that meeting project success criteria is not the same as successfully delivering business benefit. Project success criteria may actually reduce business benefit in the short term, and even more in the long term.

Type 2 Project Myopia: Beyond the project

Failure to see that successful change, transformation, services and products have a long life after the end of a project. Products and services need to continue changing if they are to succeed. Ongoing change, enhancements and renewal need to be a way of life.

Type 3 Project Myopia: Digital Business

Failure to understand that for digital business to survive and grow, digital technology needs to advance in tandem with the

business. Halting digital progress halts business progress.

Business change and transformation, product development and maintenance don't need to be set up as a project.

Prologue

> *Practical men, who believe themselves to be quite exempt from any intellectual influence, are usually the slaves of some defunct economist.* John Maynard Keynes, economist, 1883-1946

When I attended the Lean Agile Scotland conference to deliver the #NoProjects presentation I met a group of people from an Edinburgh financial services company. This group could not comprehend work without projects. Yet when I quizzed them I discovered that the same people had worked on the same software code base, on the same mainframe, to serve the same customers for over a decade. One project followed another; the only thing that was temporary about their work was the end dates.

The financial services company did not consider itself a digital company, let alone a software company, but its very ability to do business depended on information technology and software. Take away the software or finish the software and the company could not operate. It was a digital company whether it knew it or not.

At times technology management, indeed perhaps all modern management, seems to revolve around 'projects' as if they are an inherently natural phenomenon: they are not. Projects are a twentieth-century invention that has outlived its usefulness.

The project model sidelines business benefit because it chases the wrong goals in the wrong way. Traditional project planning is not a harmless, benign, pastime. It is dangerous, it reduces value and increases risk.

The project model does not describe the world of software development or the digital business world. Forcing software development into the project model requires so much mental energy, compromise, and workarounds that it becomes impossible to see what is really happening.

Managers and engineers who cling to the project model to describe their work are like aircraft ground crew that use manuals for a Supermarine Spitfire to service a Lockheed F-35 Lightning. Both are agile single-seat fighters with wings, but that is where the similarity ends.

Nowhere is this mismatch more apparent than in teams who practice continuous delivery. Such teams will inevitably abandon the project model, a model made for a temporary world.

Continuous is not temporary.

1. Introduction

> *Basic to successful project management is recognizing when the project is needed – in other words, when to form a project, as opposed to when to use the regular functional organization to do the job.* Cleland and King[1], Systems Analysis and Project Management, 1968

The simple fact is that there are projects and there is other stuff. All work does not have to be a project. However, it sometimes seems that many organizations have forgotten this fact. Projects are not the only way to organize work. Project management is not the only way to manage work.

There is work that fits the project model and can – even should – get managed as 'a project'. Then there is work that does not fit the model. Managing such work as 'a project' can hinder the work and be harmful to the final product. Software development is an example of the latter. Managing software development work using the project model makes the work more difficult than it needs to be. The project model can reduce the value of the product and result in an inferior product.

There are those who tell me that in rejecting the project model I 'fail to see the bigger picture'. In return I would say that the

[1] *Systems Analysis and Project Management*, David I. Cleland and William R. King, 1968. Excerpt taken from https://en.wikiquote.org/wiki/Project_management

opposite is true. It is because I see the bigger picture that I reject the project model. Certainly to view software development as coding alone is to ignore the 'bigger picture'. Yet to view all software development as a project – especially a defined project – is to ignore an even bigger picture.

There have long been problems in applying project management to software development. Look at the failure rate of software projects, and contrast that failure rate with the omnipresence of software-powered technology in our modern lives. Software development has succeeded *despite* the application of the project model, not because of it.

These problems have become more acute and pressing because of two forces. The first of these is the rise of agile development.

Agile software development has made the problems with the project model more apparent. As is so often the case, agile highlights existing problems and challenges workers to fix them. When agile teams are successful the limitations of the project model are even clearer. Success creates tension between agile teams and those who commission the teams.

The tensions between agile working and the project model are visible to anyone who must manage or govern projects. Successful agile teams working under the project model need to satisfy two conflicting regimes.

The arrival of *Continuous Delivery*[2] (CD) and *DevOps* added further tension. These approaches challenge the assumption of temporary built into the project model. Projects can be surprisingly long-lived, but the model used to manage them rests on the

[2] *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*, Humble and Farley, 2010.

assumption of temporary. Continuous is not temporary. CD was always embedded inside agile, but as people unlearn the project model and technology advances, CD becomes a potent force.

Reconciling the project model with agile development and CD has become more difficult, and the mismatch more obvious to practitioners. The project model itself has become an impediment to advancement.

While the incompatibility of agile working and CD with the project model makes life hard, it is a second and more important force that is challenging project thinking: the rise of *Digital Business*.

At its most crude, digital business is business based on software technology. For digital businesses to continue, to grow and to be competitive they must continue to improve the software that underpins them.

When a business is a digital business the routine work – the work the business does every day – is digital work, and digital work implies software. Improving the business means improving the software that powers the work. So improving software is itself routine work, work the business exists to do and work the business needs to be organized to do.

Businesses that end are usually seen as failures. Yet the project model regards reaching the *end* as a success in its own right. The rise of digital business makes it critical for companies to find a new model for managing and governing their technology work.

In this book I outline the problems involved in applying the project model to software development and digital business. While I make a few suggestions for how to improve things, this

book is a critique. The companion book, *Continuous Digital*[3] sets out the alternative, continuous model.

[3] https://leanpub.com/cdigital/

2. Agile project tension

To claim that projects are not undertaken in agile environments would be wrong. Agile teams undertake 'projects' all the time. Indeed, there is a large collection of literature and training that specifically discusses project management in an agile context – it's even possible to get certification in Agile Project Management.

If one examines the goals and aims of project management they can be surprisingly similar to the aims and goals behind agile. But from a common starting position the two schools develop radically different management models. Applying the project model and the agile model at the same time to the same software effort invariably creates tensions. The fundamental assumptions underlying the two models are different even if the goals are the same.

As a result proactive Project Managers and Scrum Masters devote much of their time to managing these tensions. Indeed, some executives even claim that using two models deliberately creates tension in order to control work. Yet software development and digital business are hard enough without deliberately adding complications and easy scapegoats for when things go wrong. Since the project model and the agile model have different, almost contradictory, definitions of success and failure, at some point *failure* will occur.

Many if not most teams faced with working with agile in a project model never manage or resolve these tensions: instead

they just ignore them. There are plenty of coders and testers who would not recognize the PMI or PRINCE2 definition of a project. Equally, there are plenty of Project Managers who view agile as a series of mini-projects.

Such views may well work on a day-to-day basis and serve to lessen an individual's cognitive dissonance, but they neither promote effective working nor make for good governance. Sooner or later one view comes to dominate: either the project model asserts itself and neuters agile working, or agile working renders the project model redundant. The 'projects' these teams work on are little more than accounting conventions and are a long way removed from the PMI and PRINCE2 models.

Some of the more obvious tensions between the project model and the agile model include:

- The agile model sees each small piece of work (for example a story) as a potential deliverable, while the project model aims to deliver a whole.
- The agile model optimizes for small, while the project model optimizes for large.
- In prioritizing work by business value the agile model inherently accepts that some work will fall off the end. Conversely, in the project model anything less than everything is failure.
- The agile model embraces changing requirements and specifications, while the project model aims to 'control' or even eliminate such changes ('scope creep').
- The project model expects teams to be temporary organizations, while the agile model values stable long-lived teams.

- The project model sees 'quality' as a variable one can dial up and down, while the agile model sees consistently high quality as a prerequisite for effective working and delivery.
- The project model assumes that a small number of individuals can initially scope, plan, define and design work. Then a larger group of individuals can implement the plan – *a simple matter of programming*. The agile model seeks to engage all workers in meaningful decisions about scope, plans, work definitions and design.

This list could go on. Each point of tension can be managed, but such tensions obscure what is happening and increase the mental load on workers. At some point one needs to ask "Is this model fit for purpose?"

Where the two models coexist, individuals need to engage in mental gymnastics – *double-think* – to reconcile the contradictory assumptions. In extreme cases such tensions are enough to destroy the work effort, but more commonly tension simply increases costs and risk while reducing the benefit delivered.

Sometimes tension only becomes apparent when looking across projects. For example, if an agile team is prioritizing work by business value, then one can expect a point to come at which all the high-value work is complete and the team is working on low-value requests. One might also expect their next project to contain some high-value items. The agile mind asks:

> "Why should a team undertake the low-value work to finish the first project when moving directly to the high-value work of the next project will produce a higher return?"

Even if all the work in both projects is completed one day, a cost-of-delay analysis would show that bringing forward high-value items and postponing low-value items would increase the total value delivered. On the other hand, one might ask:

> "Who really cares about the low-value tail end of a project after all?"

Fundamentally the project model defines and constrains work in order to control it, while the agile model embraces emergent needs and changing constraints. The project model is itself an attempt to constrain agility.

For non-software businesses – banks, airlines, high street retailers and so on – such tension can just about be managed. The tension might be troubling and sap energy for the workers who toil day-in, day-out inside organizations at the code-face, but away from the code-face in the executive suite things carry on much the same: project, project, project…

But not for much longer?

When the business is digital the business is a software business. *Finished* is by definition a failure.

Workers can choose who they work for, and the best are no longer choosing to work in such environments. If a company wants the best workers it can't ask them to practice double-think.

The costs of running two models, the agile model and the project model, is too high for digital businesses. The obvious costs are high already – the Project Manager and the Scrum Master, the Gantt charts and the burn-down charts – but the hidden costs are much higher:

- 'Successful' projects that fail to deliver business value.
- The reduced reactivity – *agility* – imposed by the need to shoehorn all work into a project model: to write a requirements document, design an architecture and assemble yet another temporary team.
- The lost knowledge and capability that occurs when a successful team completes the project and is disbanded. Or the hurdles and barriers managers need to jump through to keep teams together, to keep knowledge in the building.
- The loss of clarity and understanding as individuals translate from project-speak to agile-speak and back again.

Not to mention projects that fail despite all the good agile tools that get used.

A digital business can only succeed as a digital business if it is an agile business. It is no use claiming to be a digital business if it takes 27 months to introduce a new feature to match a competitor's product.

It is no longer enough for digital businesses to be 'agile in the engine room', or practice 'agile project delivery'. A digital business needs to be digital and agile all the way through.

Digital businesses that fail to utilize agile strategy and tactics will, sooner or later, meet a competitor that does. Applying yesterday's management models to tomorrow's technology only goes so far.

3. Project problems

The project model has multiple problems: it is a poor guide for managing software development work. The hashtag #NoProjects began life as a discussion of these problems. The following chapters discuss why the project model is a bad fit for software development, why it increases costs and why it destroys value.

Some offer the product model as an alternative to the project model. While the product model is a better match for software development, the two models are not symmetrical alternatives. Rather they address different issues even though they overlap. As such Project Myopia and #NoProject go beyond simply arguing that project management can complement product management.

Key to the #NoProjects critique is the observation that the development of commercially successful software is never finished. Unsuccessful software certainly finishes. When software is successful the use of the software creates the need – and opportunities – for change. Not enhancing the software hinders its usefulness. Prematurely curtailing change also curtails benefits. Therefore all successful software is a product and needs to be managed as a long-lasting product. Using the project model to deliver a series of product increments (as many development organizations do) is short-sighted and hinders value creation.

Fundamentally the project model is about temporary while successful products have longevity. This mismatch imposes costs:

management overhead, technical liabilities and, most of all, lost value. In the past companies could get away with this, but in the digital age short-sighted project thinking exacts too high a price.

3.1 Why critique?

Why write a book critiquing the project model? There are really two reasons.

Firstly, many businesses already have a pretty good way of working even with the project model. Such businesses can take a shortcut: just stop talking about projects, ban the word, and carry on working as you were but without all the project rigmarole.

There are some tell-tale signs that a business falls into this category:

- Project B follows Project A, after which comes Project C. Each project deals with the same platform/service/code-base and employs most of the same people.
- Projects start at the start of a new financial year and finish at the end of a financial year.
- You have work-steams, platforms and themes to guide your work as well as projects.

Such companies can simply start by banning the word ‘project’ and adopting new vocabulary.

In my capacity as an agile consultant, I get asked:

> “What is the hardest part of agile?”

I answer:

> "The hard bit is not what you have to learn, it's what you have to unlearn; the things you have to stop doing and the processes you need to take away."

Simply unlearning the project model is a great starting point.

The second reason for looking in detail at the failings of the project model is to learn from it. Some things within the model are good and deserve retention (deadlines, for example), but much else causes problems (temporary teams, for example.) These are things that a new model should avoid.

3.2 Projects exist

I'm not stupid: software projects do exist. I see software projects all the time. I've worked on software projects throughout my career: as a programmer, as a manager and as a consultant. I've even held the title of 'Project Manager' on occasions.

Nor do I deny that you will meet external customers and internal colleagues who want you to 'do a project'. I won't even deny that 'the customer is always right', and if the customer is offering good money for you to do a software project then your company might be stupid for turning the work down.

Most of all I am not saying the project model is everywhere and always flawed. It might be: I don't know. My expertise is in software development, the underpinning of the digital business revolution. I confine my argument to this domain; I'm happy to

speculate about extending the argument, but right now I make few claims beyond digital business and software development.

What I am saying is: the project model contains flaws when used for managing the development of software systems, especially software systems on which businesses depend. While the model can be – indeed has been – used for exactly this purpose, the flaws in the model in this context mean reduced value and magnified management problems.

When customers ask you to undertake a project you might want to consider educating them in the alternatives. Part of this education might be to highlight the difficulties of the project model.

In the short term your company may still need to undertake software projects. In the long term, your company stands to make more money and create happier customers by following an alternative model.

As the digital revolution advances, the long term gets closer.

4. Defining a project

5. Diseconomies of scale

> *Whenever a theory appears to you as the only possible one, take this as a sign that you have neither understood the theory nor the problem which it was intended to solve.* Karl Popper, philosopher, 1902-1994

Without really thinking about it, you are not only familiar with the idea of economies of scale – you expect economies of scale. Much of our market economy operates on the assumption that when you buy or spend more, you get more per unit of spending. The assumption of economies of scale is not confined to free-market economies: the same assumption underlays much communist era planning.

At some stage in our education – even if you never studied economics or operational research – you will have assimilated the idea that if Henry Ford builds a million identical black cars and sells a million cars, then each car will cost less than if Henry Ford manufactures one car, sells one car, builds another very similar car, sells that car, and continues in the same way another 999,998 times.

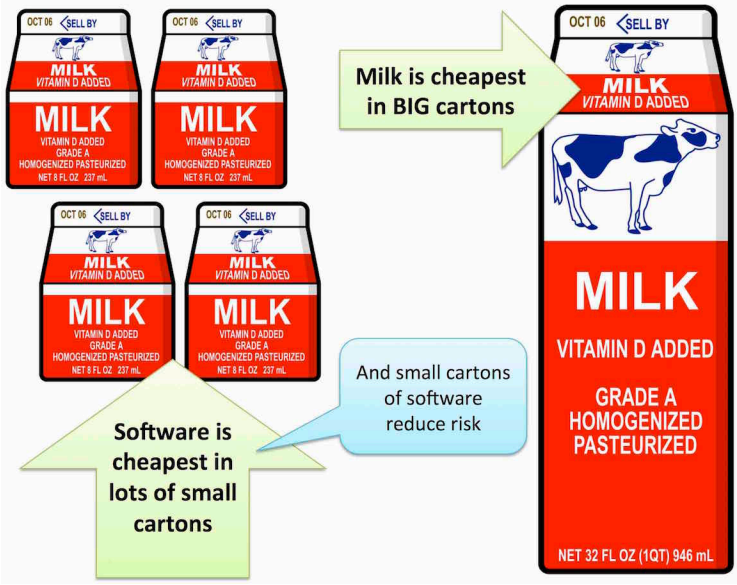
The net result is that Henry Ford produces cars more cheaply and sells more cars more cheaply, so buyers benefit. This is *economies of scale.*

The idea and history of mass production and economies of scale

are intertwined. I'm not discussing mass production here, I'm talking *economies of scale* and *diseconomies of scale.*

5.1 Milk is cheaper in large cartons

That economies of scale exist is common sense: every day one experiences situations in which buying more of something is cheaper per unit than buying less. For example, you expect that in your local supermarket buying one large carton of milk – say four pints – will be cheaper than buying four one-pint cartons.



Small cartons of software are cheaper and less risky

So ingrained is this idea that it is newsworthy if shops charge

more per unit for larger packs – complaints are made. In April 2015 *The Guardian* newspaper in London ran this story:

> *UK supermarkets dupe shoppers out of hundreds of millions, says Which?*
>
> Examples raised by Which? include Tesco flagging the 'special value' of a six-pack of sweetcorn when a smaller pack was proportionately cheaper, and Asda raising the individual price of a product when it was part of a multi-buy offering in order to make the deal more attractive[1].

Economies of scale are often cited as the reason for corporate mergers. Buying more allows buyers to extract price concessions from suppliers. Manufacturing more allows the cost per unit to be reduced, and such savings can be passed on to buyers if they buy more. Purchasing departments expect economies of scale.

I am not for one minute arguing that economies of scale do not exist: in some industries economies of scale are very real. Milk production and retail are examples. It is reasonable to assume such economies exist in most mass-manufacturing domains, and they are clearly present in marketing and branding.

But… and this is a big 'but'…

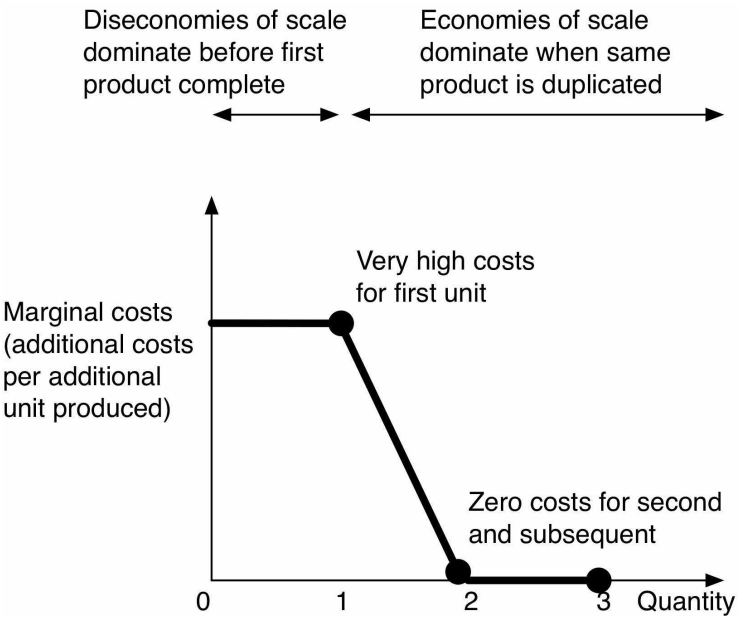
> *Software development does not have economies of scale*

[1] *UK supermarkets dupe shoppers out of hundreds of millions, says Which?*, 21 April 2015, https://www.theguardian.com/business/2015/apr/21/uk-supermarkets-dupe-shoppers-out-of-hundreds-of-millions-says-which

In all sorts of ways, software development has diseconomies of scale. If software development was sold by the pint, then a four-pint carton of software would not just cost four times the price of a one-pint carton, it would cost *far more*.

Once software is built there are massive economies of scale in reselling (and reusing) the same software and services built on it. Producing the first piece of software has massive marginal costs; producing the second, identical copy, has a cost so close to zero it is unmeasurable – Ctrl-C, Ctrl-V.

Diseconomies abound in the world of software development. Once development is complete, once the marginal costs of one copy are paid, then economies of scale dominate, because marginal cost is as close to zero as to make no difference.



Diseconomies of scale and high marginal costs give way to economies of scale and negligible marginal costs

5.2 Evidence of diseconomies

Software development diseconomies of scale have been observed for some years. Cost estimation models like COCOMO actually include an adjustment for diseconomies of scale. But the implications of diseconomies are rarely factored into management thinking – rather, economies of scale thinking prevails.

Small development teams frequently outperform large teams; five people working as a tight team will be far more productive

per person than a team of 50, or even 15. The Quattro Pro development team in the early 1990s is probably the best-documented example of this[2].

A more recent study of open source software development states that:

> 'We find strong evidence for a negative relation between team size and productivity. …we further conclude that all of the studied projects represent diseconomies of scale, exhibiting diminishing returns to scale[3]'.

The more lines of code a piece of software has, the more difficult it is to add an enhancement or fix a bug. Putting a fix into a system with a million lines of code can easily be more than ten times harder than fixing a system with 100,000 lines.

As much as software engineers love the Lego-brick analogy, software does not scale like Lego. Software exhibits power-law characteristics[4]. Some parts of the system become more central. They are connected to more parts and changed far more often. Making multiple simultaneous changes to these parts is difficult, so changes must be sequenced. Consequently bringing more people to bear on the code does not make change happen faster – it happens more slowly.

[2] *Organizational Patterns of Agile Software Development*, Coplien & Harrison, 2005.

[3] *From Aristotle to Ringelmann: a Large-scale Analysis of Team Productivity and Coordination in Open Source Software projects*, Scholtes, Mavrodiev, Schweitzer, Empirical Software Engineering volume 21 issue 2, April 2016, pre-print version available https://www.sg.ethz.ch/media/publication_files/paper_bQeEC8G.pdf

[4] *Understanding the Shape of Java Software*, Gareth J Baxter, James Noble, Marcus Frean and Ewan D. Tempero, Proceedings of the 21th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2006, October 22-26, 2006, Portland, Oregon, USA

Experience of *work in progress* limits shows that doing less at any one time gets more done overall.

Projects that set out to be *big* have far higher costs and lower productivity per deliverable unit than small systems. Capers Jones' 2008 book contains some tables of productivity per function point that illustrate this. It is worth noting that the biggest systems are usually military, and they have an atrocious productivity rate, not to mention horrendous schedule slips. The Airbus A400 transport was reportedly four years late and €5 billion over budget, while the Lockheed F35 fighter is reportedly seven years late and $163 billion over budget[5].

Testing

Testing is another area where diseconomies of scale play out. Testing a piece of software with two changes requires more tests, time and money than the sum of testing each change in isolation.

When two changes are tested together the combination of both changes needs to be tested as well. As more changes are added and more tests are needed, there is a combinatorial explosion in the number of test cases required, and thus a greater than proportional change in the time and money needed to undertake the tests. But testing departments regularly lump multiple changes together for testing in an effort to exploit economies of scale. In attempting to exploit non-existent economies of scale, testing departments increase costs, risks and time needed.

If a test should find a bug that needs to be fixed, finding the offending code in a system that has fewer changes is far easier

[5]https://en.wikipedia.org/wiki/Lockheed_Martin_F-35_Lightning_II#Program_cost_-overruns_and_delays

than finding and fixing a bug when there are more changes to be considered.

Working on larger endeavors means waiting longer – and probably writing more code – before you ask for feedback or user validation compared to smaller endeavors. As a result, there is more that could be 'wrong', more that users don't like, more spent, more that needs changing and more to complicate the task of applying fixes.

Cost of delay

Waiting is an interesting case because it has a cost. The longer it takes to deliver a product, the greater the *cost of delay*[6]. For example, the more time the product spends in development, the greater the costs, the more time it spends in development, the less time it spends in the market, the less time it is in the market before competitors arrive, and so on.

(To my mind *cost of delay* would be better called *benefit foregone* or *value foregone*.)

Those who have worked on agile teams that use small stories, or user stories, will have noticed that small stories flow through the system and are delivered sooner than large stories. For this reason agile teams often want lots of small stories rather than fewer larger stories. Unfortunately they are often met by product managers who claim that "The customer wants all or nothing. The customer will not accept anything less than everything they asked for."

[6] *Principles of Product Development*, Reinertsen D., 2009.

Cost of delay means that delivering something sooner, even if it is smaller, may well be worth more than delivering a big thing later. Even if creating the big thing enjoys economies of scale – which is doubtful – and is cheaper per unit (line of code?) than a small thing, the revenue lost because of late delivery needs to be considered.

Batch size

Software development works best in small batch sizes. There are a few places where software development does exhibit economies of scale in which case large batch sizes make sense, but on most occasions, diseconomies of scale are the norm. (Reinertsen[7] has some figures on batch size that also support the diseconomies of scale argument.)

This happens because each time you add to software work the marginal cost per unit increases:

- Add a fourth team member to a team of three and the communication paths increase from three to six.
- Add one feature to a release and you have one feature to test; add two features and you have three tests to run: two features to test plus the interaction between the two.

In part this is because human minds can only hold so much complexity. As the complexity increases (more changes, more code) our cognitive load increases, mental processing slows down, people make more mistakes and work takes longer.

[7] *Principles of Product Development*, Reinertsen D., 2009.

Economies of scope and specialization are specific forms of economies of scale and again, on the whole, software development has diseconomies of scope and diseconomies of specialization:

- Teams should focus first and broaden later when they have a working product.
- Generalists are usually preferable to specialists: technologies that demand in-depth expertise should be avoided if possible.

However, be careful: once the software is developed then economies of scale are rampant. The world switches. Software that has been built probably exhibits more economies of scale than any other product known to man.

(In economic terms the marginal cost of producing the first instance are extremely high, but the marginal costs of producing an identical copy (production) are so close to zero as to be zero, Ctrl-C Ctrl-V.)

5.3 Think diseconomies, think small

First of all you need to rewire your brain: almost everyone in the advanced world has been brought up with economies of scale since school. You need to start thinking *diseconomies of scale*.

Second, whenever faced with a problem where you feel the urge to 'go bigger', run in the opposite direction: go smaller.

Third, take each and every opportunity to go small.

Fourth, get good at working 'in the small': optimize your processes, tools and approaches to do lots of small things rather than a few big things.

Fifth – and this is the killer: know that most people don't get this at all. In fact, it's worse…

5.4 Economies of scale thinking prevails

In any existing organization, particularly a large corporation, the majority of people who make decisions are out and out economies of scale thinkers. They expect that going big is cheaper than going small and they force this view on others, especially software technology people.

Many senior people got to where they are today because of economies of scale, and many of these companies exist because of economies of scale; if they are good at economies of scale, they are good at doing what they do.

Consider banking for example. Banking, both retail and investment, exhibits many economies of scale. These occur in marketing: the same brand can offer many services and cross-sell: sign a customer for a current account, later sell them a loan, then a mortgage, then life insurance and so on.

Economies of scale occur in capital funding too. Some have even argued that size alone, while making banks riskier, also makes their funding cheaper, because governments must underwrite

'too big to fail banks'[8].

There are those who claim that modern banks are disguised software companies. Yet the individuals who reach positions of authority in a bank will do so because they are good bankers rather than good technologists. Consequently they will have spent a career exploiting economies of scale thinking. When confronted with technology concerns they will cling to what has brought success in the past: economies of scale. Inevitably this will place them in conflict when faced with a problem that requires the opposite thinking.

In the world of software development this mindset is a recipe for failure and underperformance. The conflict between economies of scale thinking and diseconomies of scale working will create tension and conflict.

5.5 And projects...

Part of the problem with projects is that they are, almost by definition, large batches of work. The administrative work involved in creating a project, getting it approved, bringing the resources together, making the resources work together effectively, then at the end unwinding all the temporary structures means that the project model only makes sense when projects are large.

Complicating matters, it can be hard to disentangle the costs of the organization from actual development costs. Some organizations demand that all work is conducted under the project model; consequently, whether the initiative is small or large, two

[8] *The Bankers New Clothes*, Admati & Hellwig, 2013

weeks of effort or two years, both initiatives require the same preparation, paperwork and approval.

In the language of economists, both initiatives have fixed costs (the start-up costs), but the longer initiative will have lower average costs. This is because the same fixed start-up costs are amortized over more production units. However, because the larger initiative requires more coordination, the marginal costs per unit will be higher. Consequently it can be hard to do true cost comparisons between endeavors.

Companies seem to like projects: projects imply change, and change implies growth. This is much more attractive than 'business as usual'. But the need for projects to be large means small is not an option, and therefore the stakes are high and the risks are large.

Unfortunately, software development lacks economies of scale. Time and time again building software in the small is more efficient than doing so in the large.

Software is cheapest in small quantities.

There is an inherent conflict between the best way of running a project and the best way of organizing software development endeavors.

5.6 Making small decisions

In part big-batch projects are an attempt to maximize the value of the most precious limited resource: senior management time. Getting time with a senior manager is difficult, their interest

in discussing anything worth less than $10 million is negligible (replace with a relevant figure for your organization). So why bother them with small pieces of work? You are more likely to get a few minutes of their time to approve a $10 million project than to discuss 100 small $100,000 pieces of work.

For software development to exploit the rampant diseconomies of scale, decision authority needs to be devolved downwards so that small decisions can be made efficiently when needed, rather than bundled into single big decisions.

5.7 Optimize for small

Diseconomies of scale mean that organizational structures need to be reconsidered. Individuals, teams and organizations need to learn to think small. They need to start looking for *small*:

> *Teams need to organize themselves for lots of small.*
>
> *Organizations, teams, processes and practices need to be optimized for small.*

Because of diseconomies of scale, it is necessary to rethink the traditional economies of scale-based organizational structures, and create structures and processes that are optimized for small. Only by optimizing for small can organizations and team exploits diseconomies of scale.

Work processes need to be optimized for small pieces of work – small batch sizes and small items. This means big activities (for example set-up, teardown, one-off reviews) need to be removed.

Things that are expensive and get minimized (such as sign-off and final test cycles) need to be removed or rethought so that they can be efficient in the small.

Each of those small pieces of work needs to demonstrate potential value and be evaluated later for value delivered.

5.8 Kelly's Laws

I have two personal laws.

Kelly's first law of project complexity

> *Project scope will always increase in proportion to resources.*

The more people, time, and money you have, the more your project will attempt to do.

Kelly's second law of project complexity

> *Inside every large project there is a small one struggling to get out.*

Look for the small piece of work struggling to get out, then work to deliver that early.

6. Software isn’t temporary

7. If they use it, it will change

The most loved and legendary building of all at MIT is a surprise: a temporary building left over from World War II without even a name, only a number: Building 20. …constructed hastily in 1943 for urgent development of radar and almost immediately slated for demolition. Stewart Brand[1], author

Building 20 was demolished in 1998, 55 years after its construction.

[1] *How Building Learn*, Stewart Brand, 1994

8. False projects

9. The problem with project success

10. Multiple projects

11. Increasing value

12. Debt thinking

13. The quality problem

Defects are not free. Somebody makes them, and gets paid for making them. John Cage, composer, 1912-1992

The bottom line is that poor-quality software costs more to build and to maintain than high-quality software, and it can also degrade operational performance, increase user error rates, and reduce revenue by decreasing the ability handle customer transactions or attract additional clients'.

For the software industry, not only is quality free, as stated by Phil Crosby, but it benefits the entire economic situations of both developers and clients.* Jones, 2011[1].

[1] *The Economics of Software Quality*, Jones, C., Bonsignour, B. and Subramanyam, J., 2011

14. Programmes not projects

15. Personal changes

Do not think of today's failures, but of the success that may come tomorrow. You have set yourselves a difficult task, but you will succeed if you persevere, and you will find a joy in overcoming obstacles – a delight in climbing rugged paths, which you would perhaps never know if you did not sometimes slip backward – if the road was always smooth and pleasant. Helen Keller, author and activist, 1880-1968

Want to read more?

I hope you enjoyed this sample. If you would like to read more…

Buy Project Myopia today on Amazon[1]

[1] https://amzn.to/2wZW9JM

Continuous Digital

Continue the #NoProjects story with Continuous Digital continues

Allan Kelly's latest book:

- Why digital business need a new model of software development
- A full description of the Continuous model

Ebook draft available on LeanPub[2] and pre-order on Amazon[3].

[2] https://leanpub.com/cdigital/
[3] https://www.amazon.co.uk/Allan-Kelly/e/B001JSFJEE

About the author

Allan inspires digital teams to effectively deliver better products through Agile technologies. These approaches shorten lead times, improve predictability, increase value, improve quality and reduce risk. He believes that improving development requires broad view of interconnected activities. Most of his work is with innovative teams, smaller companies - including scale-ups; he specialises in product development and engineering. He uses a mix of experiential training and ongoing consulting.

He is the originator of Retrospective Dialogue Sheets[4], the author of several books including: "Xanpan - team centric Agile Software Development" and "Business Patterns for Software Developers", and a regular conference speaker.

Contact: allan@allankelly.net

Twitter: @allankellynet[5]

Web: http://www.allankelly.net/[6]

Blog: http://blog.allankellynet/[7]

[4] http://www.dialoguesheets.com/
[5] https://twitter.com/allankellynet
[6] http://www.allankelly.net/
[7] http://blog.allankellynet/

Also by Allan Kelly

Little Book of Requirements and User Stories

Available from your local Amazon[8]

Xanpan: Team Centric Agile Software Development

Ebook: https://leanpub.com/xanpan[9]

Print on demand: Lulu.com[10]

And your local Amazon[11]

Business Patterns for Software Developers

Published by John Wiley & Sons

Available in all good bookshops and at Amazon[12]

[8] https://www.amazon.com/Little-Book-about-Requirements-Stories-ebook/dp/B06XZZ6BQD

[9] https://leanpub.com/xanpan

[10] http://www.lulu.com/shop/allan-kelly/xanpan-team-centric-agile-software-development/paperback/product-22271338.html

[11] https://www.amazon.com/s/ref=nb_sb_noss?url=search-alias%3Daps&field-keywords=Xanpan

[12] https://www.amazon.com/Business-Patterns-Software-Developers-Allan-ebook/dp/B007U2ZT7K

Changing Software Development: Learning to Be Agile

Available in all good bookshops and at Amazon[13]

[13] https://www.amazon.com/Changing-Software-Development-Learning-Become/dp/047051504X