

**Batch size  
matters**

This publication forms part of Value, Flow, Quality® Education. Details of this and other Emergn courses can be obtained from Emergn, 20 Harcourt Street, Dublin, D02 H364, Ireland or Emergn, 190 High St, Floor 4, Boston, MA 02110, USA.

Alternatively, you may visit the Emergn website at <http://www.emergn.com/> education where you can learn more about the range of courses on offer.

To purchase Emergn's Value, Flow, Quality® courseware visit <http://www.valueflowquality.com>, or contact us for a brochure - tel. +44 (0)808 189 2043; email [valueflowquality@emergn.com](mailto:valueflowquality@emergn.com)

Emergn Ltd.  
20 Harcourt Street  
Dublin, D02 H364  
Ireland

Emergn Inc.  
190 High St, Floor 4  
Boston, MA 02110  
USA

First published 2012, revised 2021 - printed 16 July 2021 (version 2.0)

Copyright © 2012 - 2021

Emergn All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, transmitted or utilised in any form or by any means, electronic, mechanical, photocopying, recording or otherwise, without written permission from the publisher.

Emergn course materials may also be made available in electronic formats for use by students of Emergn and its partners. All rights, including copyright and related rights and database rights, in electronic course materials and their contents are owned by or licensed to Emergn, or otherwise used by Emergn as permitted by applicable law. In using electronic course materials and their contents you agree that your use will be solely for the purposes of following an Emergn course of study or otherwise as licensed by Emergn or its assigns. Except as permitted above you undertake not to copy, store in any medium (including electronic storage or use in a website), distribute, transmit or retransmit, broadcast, modify or show in public such electronic materials in whole or in part without the prior written consent of Emergn or in accordance with the Copyright and Related Rights Act 2000 and European Communities (Copyright and Related Rights) Regulations 2004. Edited and designed by Emergn.

Printed and bound in the United Kingdom by Apple Capital Print.

# CONTENTS

## **Introduction 1**

### **1 Batches 2**

1.1. What is a batch? 2

### **2 Batch size in software development 4**

2.1. Big batches in software 6

### **3 The benefits of small batches 8**

3.1. Reduced cycle time 8

3.2. Faster feedback and reduced risk 10

3.3. Reduced overhead and increased efficiency 13

3.4. Heightened urgency 13

### **4 How to create small batch sizes 14**

4.1. Deciding on a new batch size 14

4.2. How do we make our batches smaller? 15

4.3. So why do we still have large batches? 24

### **5 The trouble with I.T. 27**

5.1. 'Ah yes, but THIS project is different' 27

### **6 Conclusion 29**

### **Bibliography 30**



# INTRODUCTION

In this session we will examine the concept of batches and how their management impacts on queues and cycle time. This is a subject area which many software development teams are not conscious of measuring or managing, despite the fact they will inevitably come across methodologies that focus on batch control (e.g. Scrum). We will examine the dangers of large batches and the benefits of small batches. We will also look at why large batches have a tendency to occur – despite their disadvantages. When considering how to reduce batch size, we will also run through the likely obstacles, specifically transaction costs, but also structural issues within a company as a whole.

At the end of this session, the student will be able to:

1. Understand what a 'batch' is in the context of software development.
2. Appreciate why small batches lead to fewer queues and faster cycle times.
3. Recognise that small batches also decrease risk through faster feedback, while increasing urgency.
4. Understand the relationship between batch size and transaction cost.
5. Identify ways to reduce batch size by time, input or size.
6. Recognise a natural bias for 'large' projects that leads to large batch sizes.
7. Identify the likely catalysts of large batch size within software development processes.

# 1 BATCHES

In the 19th century, London's population was exploding. New housing was desperately needed. Farmland was bought, divided up into plots and houses were built. The rows and rows of terraced housing that make up much of Tottenham, Hackney, Catford, Deptford and other boroughs were the result. The surprise to some is that they were built, not by an enlightened government or local councils, but by small, speculative builders. A builder, normally just one man with perhaps a carpenter for help, would put up 2 or 3 houses with an identical plan and then stop. After the houses had been sold, he would invest the profits in erecting the next few. You can still spot where each spurt of activity lasted. Brick colour may be different, chimney pots suddenly change shape, a new decorative motif... These builders were working in small batches, and the story should give a clue to the benefits: small capital outlay, low risk, short cycle times.



Figure 1. Non-identical houses in Hackney

## 1.1. What is a batch?

It's perfectly acceptable to look blank at this point. In our experience, not many software developers or IT managers have heard the word in a development context, despite the fact that batch size reduction has made some of the biggest advances in improving working methods and productivity. You have probably worked in places which use these methods (e.g. prototypes, project phases, agile). Without an understanding of why, it is difficult to apply the proposed guidelines effectively or to extract the maximum benefit.

Think of a batch as either an idea or a quantity of work, required for or produced by, a single operation. You can process work or ideas either in large quantities or small. The quantity is the batch size. The idea comes from manufacturing, when the batch would be the amount of goods produced at one time, but has been adopted by software development.

David Anderson neatly encapsulates the key point as this:

'Finished code cannot be delivered without each idea being processed through a series of transformations. Whether or not the ideas are all processed together (waterfall – big batches), or they are processed in small batches (almost all forms of Agile)... the number of ideas processed is bounded (by time, or size)."

---

### **Activity 1: Batches in your workplace**

This activity will take approximately 15 minutes.

In a typical software development process, batches abound; whenever items are packaged together, we see a batch created. Examples include: the funding process that will only consider projects that deliver a defined amount of value and consequently, the number of requirements that are packaged into a project; release processes that wait until all requirements are completed before they are conducted; the number of User Stories that are taken into a Sprint.

Gather some of your colleagues together and explain to them what a batch is.

Now, in the second half of the activity, consider the development process within your own organisation and how the ideas that are generated flow through to the customer. Discuss batches that exist within your process and note these down, as we will use them in the next activity.

## 2

## BATCH SIZE IN SOFTWARE DEVELOPMENT

Software development has traditionally encouraged large batch sizes: ‘Build me an ERP system’; ‘The customers need a new website’; ‘We need to do our annual release in August’.

A rule of thumb in our heads insists that this is more efficient because we can do everything at once, rather than piecemeal. It makes sense to ask the customer to review all the designs in one go; it’s simpler to have a tester work on the entire system once it’s finished; we should search for bugs once we’ve written the code; there’s no point in showing marketing the product until the user interface is done...

Suppose we had a pile of books to move down stairs: we could pack all the books into one very large box and move the box, or we could carry each book individually downstairs. The second option sounds stupid, right? Think of all those wasted trips!

But large batches come with their own problems. The box into which we pack all our books turns out to be very heavy and hard to manoeuvre. In fact, to use it properly we end up having to build a special reinforced crate and a hoist and tackle to lift it down the stairs. Working all this out takes us a terribly long time. Building and ordering the box might take even longer – so long that carrying the books down the stairs one by one starts to look like the sensible solution.

Between these two extremes of 100% batch size, and granular batch size, there is an alternative (and for all the software developers reading this, the alternative is not to download all the books onto your Kindle and just carry that down the stairs ... so stop being so clever). The alternative is to pack a few books into a smaller box and carry this down the stairs. Was it easy? Maybe the next time we’ll stick a few more books in. Too hard? Next time we’ll do it with fewer.

This rather simplistic example introduces the central benefits of changing batch sizes: it is cheap, reversible and encourages a fast feedback loop. We will go on to show you that batch size reduction manages to reduce queues and therefore cycle time without changing either capacity or demand. This makes it a powerful tool for software development.



Figure 2. Books on stairs waiting to be moved

---

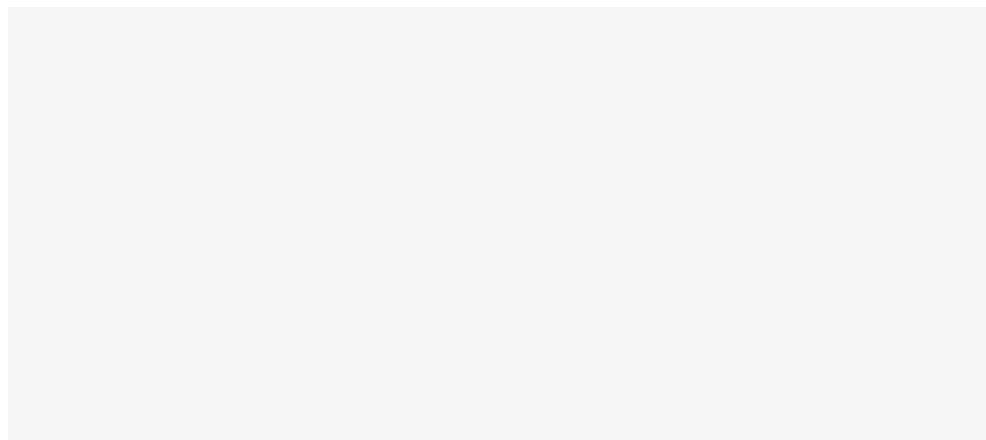
## **Activity 2: How big is your batch?**

This activity will take approximately 15 minutes. You can do it either on your own, or with the same group of colleagues from the first activity.

Take the list that you created in Activity 1. Pick five of the batches that you know well.

Now try to estimate their size - that is, how long would each one take to complete.

To help with this, remember that in software development we're working with ideas. When estimating your batch sizes think about how these ideas are represented. How many do you deal with at once?



### **Commentary:**

In our experience software development departments often work with very large batch sizes although sometimes this may not be immediately obvious. The testing stage, for example, may deal with 20 test scripts at a time. Each script takes only 5 minutes which you might calculate as 100 minutes. This will not seem like a large batch size unless you consider the number of requirements each script is testing, perhaps as many as 20. This would give a total of 400 variations, or 2,000 minutes - a significant amount of work. To get a realistic estimate of the batch sizes your process deals with it's best to look at the inputs.

---

## 2.1. Big batches in software

Software development abounds in examples of huge batches. One common example is when we insist on a complete product specification before design begins – it encourages huge numbers of features, because at this stage customers add in anything they might possibly want and the result will be a correspondingly long development time. Releases are another classic example – companies release a huge number of upgrades to a product in one go, saving changes up until they can all be launched together.

Reviews or stage gates have a similar function. Although they are intended to reduce a company's risk through due diligence, their effect is often to increase it because they lead to large batches. Since most stage gates demand that everything from an earlier stage must be completed before the project can continue, the batch size at this point is at its maximum.

At the beginning, an idea is presented requesting approval, which normally releases large amounts of funding. Since a lot of work must go on before this in order to justify the funding (business case, design specification, risk analysis etc.) teams tend to ask for anything they might want, in order to avoid having to come back to ask for a little bit extra later on. Thus, before the project has even begun, it is essentially set in stone. Only a very few projects can be admitted to the development process (cutting out lots of good ideas) and these quickly become "too big to fail".

The problems continue throughout the process, with the gates requiring a full completion of each stage before the next begins. Delivery time is slower since long queues form downstream. The whole batch becomes dependent on its slowest element, when actually many of the activities could have been run concurrently reducing queues and improving flow.

## CASE STUDY: Microsoft Windows 7



Figure 3. Microsoft Windows 7 with SP1 packaging

Really? Is it that time again? Didn't we do it last week?

Nobody (with the exception of a few geeks who enjoy paying to work as testers) likes having to upgrade their system. Companies in particular hate having to do it. Their misery is less to do with the cost of a new license, and more about the potential risks associated with large batches.

When Microsoft launched Windows 7, 60% of all respondents to a ComputerWorld survey said that they would wait to adopt the system, and 65% said they would wait at least until Windows 7 Service Pack 1 was released. Why?

A major upgrade represents a large batch. It has major associated risks – the most important of

which is probably security. What might go wrong when the company switches computers on their network over? What systems might fail? What crucial projects might be endangered? Never mind what the licenses cost, what other costs will be incurred to fix errors or security breaches?

Of course, Microsoft markets the upgrade hard. There are a host of benefits to be had from switching: faster, better, fancier... The fact is that such promises do not outweigh the cost and risk. That's why Microsoft has to force its customers to switch by refusing to support old operating systems after a certain point.

Customers accept that this business model exists and they will have to switch eventually. But almost all of them will wait until the Service Pack is released – which will fix the errors and holes in the original release. The difference between a big batch and a big batch +1 is very small, perhaps even negligible, so far better to wait until some early adopters have taken the risk and discovered all the problems in Windows 7 for you.

The irony is that the decision-makers over Windows 7 adoption who have correctly identified the risks connected to large batches, are the very same people who are producing their own software in large batches. Presumably they feel that their own software will have fewer bugs and errors than Microsoft's or that customers will buy their marketing claims (even though the CIOs didn't buy Microsoft's)... It's a comfortable conviction that doesn't bear much scrutiny.

Microsoft adopted the large batch size because this is what makes the financial model of upgrade with new license more marketable. Interestingly the idea in the industry is gaining ground that a subscription model with continuous upgrades (small batches) may prove far more attractive to companies and be just as advantageous financially.

# 3

## THE BENEFITS OF SMALL BATCHES

### 3.1. Reduced cycle time

That sounds like a bit of a magic trick. To understand it, think of a traffic jam caused by two lanes narrowing to one. If 100 cars arrive all at once, there will be a long queue. If 100 cars arrive in a spaced out fashion, say 1 every 15 seconds, the cars will pass through the constricted point at a steady speed that will be faster overall. This is the power of batch size to optimise flow.

One methodology designed to reduce batch size and optimise cycle time is Scrum. This limits batch size by time, insisting on developers committing to tasks which can be completed within 30 days (nowadays often reduced to 14 days). By the end of the 30 days, a working piece of software has been delivered.

The big problem or project still exists, but by breaking it down into smaller batches (each of 30 days), we reduce the variability of flow. If we have deviated from the plan we have an early indication that we are doing so because we receive feedback at the end of the Sprint. Now instead of the 100 cars arriving at the constriction all at once and forming a queue, we have limited the cars arriving. Their steady rate reduces variability and stops queues forming.

Scrum and other agile methodologies have been extremely effective in speeding up cycle time overall, and they have related benefits as well. However, it is fair to say that breaking a problem into similar sized pieces of work is not always easy. A reasonable criticism of Scrum is that the very nature of software development or IT is that large batches do arrive, often very unpredictably and at high impact.

Our example was 100 cars – these separate obviously into 100 individual cars. IT batches tend not to be quite so easy. ‘Change all our systems within the year’ is a directive that might arrive from the board. As a batch, it’s huge! It can be split down – but doing so will take time, and the Scrum framework itself does not help you to solve this problem.



Figure 4. Two lanes merging

**CASE STUDY:****Swedbank Arena: Solna, Sweden**

Figure 5. The new Swedbank Arena in Sweden

The Swedbank Arena is a new stadium hosting football matches and concerts, with seating for up to 65,000 people, which opened in 2012. While it comes with all the features you would expect – bars and shops, huge digital screens, sliding roof, choice of grass, ice or wooden flooring underfoot – the most important factor is the attention that has been paid to crowd flow and dynamics. The Swedbank Arena website proudly claims that the entire stadium can be emptied in 6 minutes.

When people arrive at a large stadium they come in convenient small batches. They may be arriving by car – in which case you have a steady stream of batches of 4 people or fewer. They may arrive by train, in which case the batch sizes will be larger, but still a steady flow of groups of about 100 or so.

The real issue comes when people are leaving the stadium. At this point you have a very large batch size of 65,000 all trying to leave at the same time. Swedbank Arena employed an expert in flow – Crowd Dynamics – to assist them with this. The work includes both increasing capacity (opening up various extra exits to get people out quickly) and facilitating the flow to get people to these exits. A crucial feature is also to artificially limit the batch sizes. If overcrowding begins to occur, staff hold people at preselected locations, releasing small groups slowly into the flow.



Figure 6. Open walkways in the stadium

## 3.2. Faster feedback and reduced risk

Just to return to our abstract book-moving example. If we were professional book movers, it might be worth building reinforced boxes and setting up hoists, because we are pretty sure that we will be moving a similar amount of books tomorrow. In IT and software development, we have few predictable, identical problems. Instead it is more likely that tomorrow's project will involve a completely different solution. Software development constantly needs to try out new ideas and solutions. That means there is a huge economic value to gaining feedback on these. Is our solution working? Is it what the customer wants?

Effective feedback is driven by small batch sizes. Small batch size provides faster, more regular and early feedback, all of which reduces risk. Again Agile methods encourage this. They insist upon feedback, either directly from external customers or by working with business users, who see a piece of working software demonstrated at the end of each Sprint. It means we can test a solution – fail – and develop a new one. If we had spent months and months on a solution, this would be extremely costly. Just imagine if we built our reinforced crate, set up the tackle and then discovered it still wasn't strong enough... we've spent effort and money and we still have an untidy pile of books! In software, not only is there a risk of failure, but our ideas may turn out to be less valuable than we had thought (as time passes).

Gaining early and regular feedback enables us to change direction easily and cheaply. Releasing a product quickly follows the same principle – by generating real feedback from the customer (both sales and anecdotal) we can see our mistakes, allowing us either to fix them or even kill the project in order to cut our losses. By generating positive feedback around specific features, it also tells us where to invest the most effort for the next release.

A short feedback cycle powered by small batches means the organisation, or product, can remain flexible and nimble. Agile is based on this concept. The clue is in the name.

We mentioned that Scrum cycles have reduced from 30 to 14 days. The benefit is less about decreasing cycle time, than limiting the batch size to get earlier feedback. In 30 days too much work can be done that puts at risk the principles of early rapid feedback, increased learning efficiency and greater motivation and urgency. Limiting the batch size makes the project more flexible because feedback is incorporated more frequently.

This does not only apply to large projects or products, it is also directly applicable at the idea development or start-up phase of an organisation. Lean Startup encourages developing the 'Minimum Viable Product' (MVP) to ensure the feedback loop is as fast and effective as possible. The MVP is not necessarily the smallest or simplest product imaginable, but it does mean ruthlessly excluding any feature, process or effort that does not directly contribute towards providing the learning you need.

## CASE STUDY: IMVU



Figure 7. IMVU 3D avatars

When Eric Ries began the startup business IMVU, he and his colleagues gave themselves a stretching target – launch a product within six months. As the launch date approached, the team became increasingly desperate. Eric Ries writes, ‘I won’t mince words, the first version was terrible’. It had hundreds of bugs, they had to leave

out lots of features which they were convinced were essential, they were worried about damaging the brand and on a personal level they were worried about their professional reputations... And yet, they managed to hit the launch date and ship a product.

### What happened?

Nobody even noticed the defects, because no-one bothered downloading it.

Eric Ries and his team had a long journey towards discovering what they had done wrong and which assumptions were flawed. It involved talking to hundreds of customers, observing what they did, how they used the product and creating dozens of different versions along the way. None of which would have been possible without real market feedback. As Ries points out – focus groups and research were of limited use to a company launching something that was entirely new. Nobody knew whether they wanted a 3D avatar yet. The only information that really helped was market feedback – customers’ action or inaction in the real world.

If the team had delayed the original launch in order to ship the perfect product, they would have found out much later about the huge flaws which underlay their development. That delay might have proved the difference between being able to rescue the project, and going bust. Ries feels he should have shipped a product sooner – a lot less effort could have been expended in order to achieve the same learning (that their strategy was flawed). This gives an important insight – no matter how good you think you are being about shipping the MVP – you can almost always do better. You need to be rigorous in asking yourselves what learning you will achieve through shipping NOW.

### **Activity 3:** What is your Minimum Viable Product?

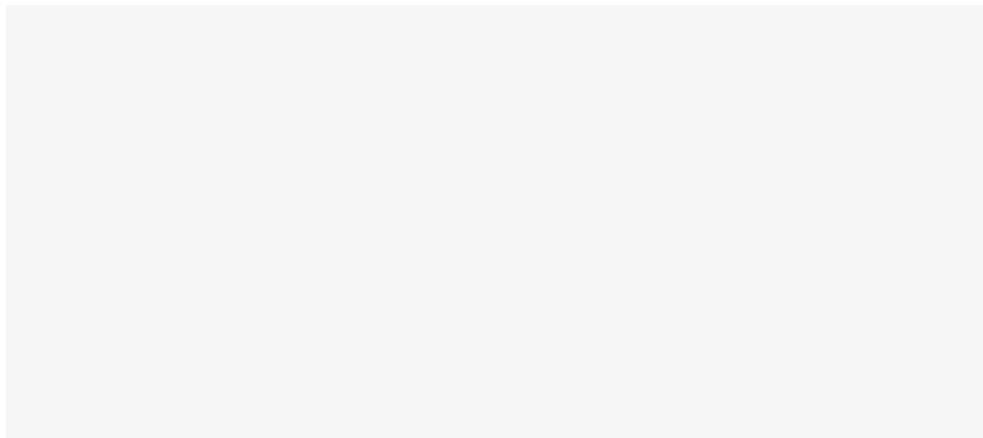
---

"As you consider building your own minimum viable product, let this simple rule suffice: remove any feature, process, or effort that does not contribute directly to the learning you seek."

This is how Eric Ries describes MVP in his book *Lean Startup*. Take this approach and see how it applies to a product you're currently working on.

This activity should take about 10 minutes.

List all the features that are currently being worked on for the product you have selected. What are you hoping to demonstrate or learn at the new product demonstration? Consider what features you could remove and still obtain useful feedback?



#### **Commentary:**

This approach may seem more applicable, for example, to a startup environment where we want to validate our idea with the customers before building the product. It can also be useful as we iterate an existing product. When enhancing a product, for example, using Scrum, you might be hoping to find out whether the new feature will meet the needs of your users. In this context, you can consider how much of the feature's functionality is required before the users can confirm that it will meet their goals.

---

### 3.3. Reduced overhead and increased efficiency

If the power of small batches to reduce risk is difficult to grasp, their capacity to reduce overhead feels completely counter-intuitive to many developers and managers. It was a surprise discovery to manufacturers when they began to implement lean methods and small batch sizes in factories, but all the evidence suggests that exactly the same is true for software development.

It works for two reasons. The first is that the time it takes to perform a task is not necessarily fixed. It takes us longer to put up a shelf the very first time we do it than the second time. By the time we are putting up our tenth shelf, we have become far more efficient through our ability to learn.

The second reason is connected to smaller queues. Greater gains are made by optimising the efficiency of the whole, rather than any one step. Focusing on what appears to be an efficient way of working – completing all the design up front, which releases designers to work on something else – actually creates a big inventory of partially done work. This large batch hides a multitude of potential errors from design to integration. There is also a far greater need to track this inventory. If you have lots of bugs open, each new bug must be checked against the list of known errors, and there will probably be status reports generated in order to keep track of progress – all of which adds to overhead costs.

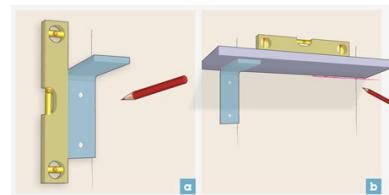


Figure 8. How to put up a shelf!

### 3.4. Heightened urgency

"If it's going to be a long time before a customer sees a story in the middle of the queue, there is little sense of urgency. But if the customer is going to see it soon, we better worry about getting it right, right now."

Dean Leffingwell, Agile Software Requirements

Used correctly, short deadlines drive urgency. A short cycle encourages responsibility and motivation. Developers who know their code won't be incorporated for several months have less motivation to be rigorous in checking for errors – it doesn't mean they're bad people, just human.

# 4 HOW TO CREATE SMALL BATCH SIZES

## 4.1. Deciding on a new batch size

A new rule of thumb:

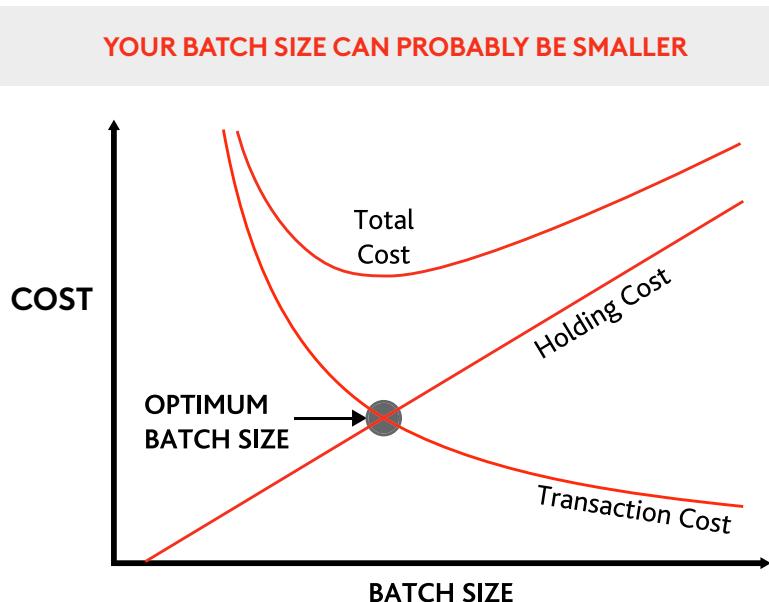


Figure 9. Optimum Batch Size (taken from The Principles of Product Development Flow by Don Reinertsen)

Optimum batch size exists at the point where the cost of holding onto the batch meets the cost of transaction.

Doesn't it look great? You can certainly impress your colleagues with a whiteboard and some pens at the next meeting as you plot your own figures on the graph to calculate the optimum batch size. Or you could just remember the two things we told you at the start of this section: batch size reduction is cheap and reversible.

Note that the optimum batch size graph is a U curve – that means it is what Reinertsen calls ‘very forgiving of errors’. If you are one of the very rare companies operating at the optimum batch size right now, you could reduce batch size further without suffering a monstrous cost increase. If, as is far more likely, your batch sizes are too large, you can reduce them step-by-step and measure the results. This also means you can continually change batch size in response to changing conditions. For example, on a very innovative, unusual project, you might want to meet with the customer every week, following a one-week iteration. As these components are locked down and you move on to other, more utilitarian ones, customer approval meetings could reduce in frequency as the iterations lengthen, first to 14 days and then to 30.

Reinertsen points out that most companies underestimate the cost of not releasing value. Once again, this is because it is intangible. If you had a pile of half-made handbags lying around, the cost of storing them would be one you were painfully aware of. The thought that they might go out of fashion if you didn't ship them soon would haunt your dreams. Software may have a longer shelf-life than fashion handbags... but only just.

If the figures going into your calculation and graph are flawed, why bother? Instead test a new rule of thumb: your batch size could probably be smaller than it is. So reduce the batch size, and if transaction costs jump, reconsider.

Please note that we don't say: 'if it doesn't work, reconsider'. Batch size reduction is not necessarily easy and it needs to be supported by changes beyond software development.



Figure 10. Gucci storing unfinished handbags on the wall

## 4.2. How do we make our batches smaller?

When discussing traffic flow, we pointed out that each individual car is a small, neat batch. Annoyingly, software problems aren't, nor do they come ready marked as to how they should be broken up, like the squares on a chocolate bar.



Figure 11. Squares on a chocolate bar

The first way to limit batch size is to limit what we put into it. For example, we might limit the number and size of requirements that we are grouping together. Scrum is a classic example of this, taking on only as much work as can be completed within the Sprint.

Not only is limiting the input the simplest method – it's also the cheapest. Most companies can make huge advances in cutting their cycle time simply by focusing on

what they put into the batch in the first place. This approach does not have any significant effect on overall cost. Eventually, however, while we see the benefits of smaller batches in reduced cycle times, we begin to creep along the U curve of the graph – the batch size may be getting smaller, but the transaction cost will eventually start to rise. At this point the forward-thinking company can use some of the money saved through shorter cycle times to reinvest in lowering transaction costs.

Traditional manufacturers who were considering the ideas of the ‘Lean’ revolution that Toyota had pioneered scoffed at first. The rule of thumb was that small production runs (small batches) have a high transaction cost. Changing over the ‘die-stamping’ machines in American factories took 24 hours. That made it madness to consider running the machine for just a few hours before changing the dies again in order to produce something else. The brilliant Japanese manufacturing engineer Shigeo Shingo, examined the changeover time and reduced it. Then he reduced it again. Eventually the changeover took only 10 minutes.

This thinking applies equally to software development. People believe that if they do something more frequently they will have to pay for it more often, raising total costs. They think of the transaction cost as fixed and therefore reject small batches out of hand.

Just as Shingo invested time and effort in working out how to reduce the changeover time, so we can reduce software development transaction costs. Each case is different, which makes it hard to give a recipe for success. One of the things Shingo proved was that if you believed that a fixed cost could be reduced, then it really could be. Often, reducing transaction costs means investing in infrastructure and automation. This is especially important because the frequent feedback cycle will mean we are likely to be retesting the increment as we make changes to it.

Upfront investment can even stretch to the way we set up our meetings – short regular meetings have a low transaction cost compared to large one-off approval boards, for example. Many projects end with a review – an examination of what worked and what didn’t and a series of recommendations for the future. These review documents often end up on a shelf gathering dust. Scrum, by contrast, insists on review meetings at the end of every Sprint, this means that what worked and what didn’t work is a dynamic action list, which tries to ensure that lessons are applied as they are being learned.

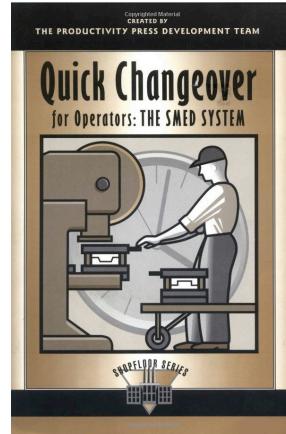


Figure 12. Shigeo Shingo’s book, Quick Changeover

**CASE STUDY:****Wiredreach moves to continuous delivery**

Wiredreach is a small company which writes software to enable customers to share information securely from their desktop. Founder and CEO, Ash Maurya, used to release updates on a weekly basis. As he said, he had always felt that this was, "pretty agile, disciplined, and aggressive". The release took time, however, – up to 20% of the working week – a process which seemed very wasteful for a small team. Eventually, competing pressures between developing software for new customers and remaining responsive to existing customers (with the regular weekly release) began to build. At this point, the team had a choice – to move to less frequent releases, every two weeks or every month; or to move to continuous delivery (continually releasing software). They decided that continuous delivery was the route they wanted to follow – even though releasing without a safety net felt rather frightening.

It meant that the team needed to invest some upfront time in automated processes that would help them. They built functional tests and identified events that would trigger an alarm should something go wrong (no users on the system etc). Automating testing and monitoring in this way allowed them to release with more confidence. Inspite of tests, bad code can still be released, but the speed of the feedback loop means that the error is identified quickly and can be instantly traced to source because there's no need to search through a large batch. Customers have learned to appreciate a 'near-instant' fix to problems. Moreover the team keeps things simple by deleting features that aren't being used – meaning there is less code to maintain.

Note that the eventual time-saving of 20% per week from the original release schedule, could not have been realised without investment. This should not just be seen as a one-off. Continuous improvement of the process is a better use of that time, but there is a danger that a short-sighted view might overload the team with extra work in the belief that they now have more capacity.

## Reduce dependencies

Creating small batches is not always easy and it can require changes beyond software development. This is why Agile is sometimes called pervasive, it goes further than merely making changes to software development. To benefit fully, it requires changes in the wider organisation.

Small batch sizes require the breaking down of dependencies, but dependencies are common, and hard to tackle without significant will and investment. To take a technical example, if we wish to test sub-systems before the system as a whole is ready, we need to make the sub-systems independent and we need to create the infrastructure which permits us to do this. On a more general, company-wide example, if we wish to produce regular, frequent releases, marketing, sales and senior management (or whoever gives final approval) also need to be organised to support such a schedule.

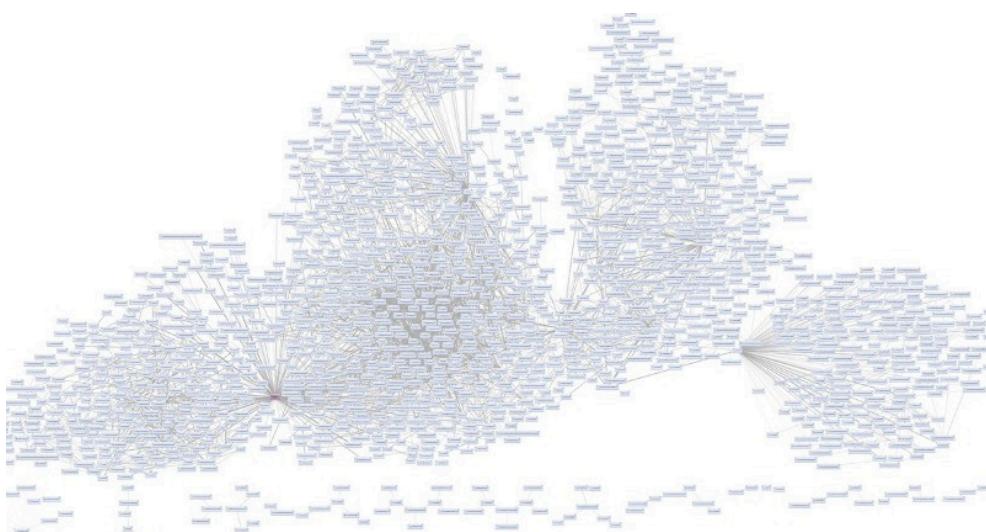


Figure 13. Legacy System Dependency Map – Note that this architecture is less than four years old and contains 17 times as many classes as the previous legacy system – legacy system does not mean old

Legacy systems are especially likely to suffer from dependencies. Make a single change to one part of the system and hundreds of changes might be required as a result elsewhere. Since these normally come with technical constraints, the result can be a major headache. Improvements can be made to batch size without affecting the system architecture, but once companies begin to look at reducing their batches to the size required by continuous delivery, for example, it becomes necessary to ensure the system as a whole supports the changes.

Changing the system by reducing these dependencies tends to require significant investment. In order to build an effective business case to make such changes, you need to examine current costs of delay that are caused or exacerbated by large batch sizes. When these costs are set against the cost required to re-engineer existing architecture, you can make an informed decision about whether to begin such a project.

There is an implicit and important relationship between the structure of what we build (our architecture) and the way we work. Service Oriented Architecture and other models offer promises to deliver improvements in how fast and easily we can deliver software. However, since most of our delays are caused, not by the development process itself, but by queues, the most perfect architecture imaginable will be unable to make a difference, unless it is combined with changes to the way we work, including enabling development processes through small batches and limited work in progress (WIP).

A similar cost benefit analysis is required when deciding how to make changes to company processes. Small batches require a more frequent approval process, whether for funding or prioritisation of features. If these decisions have to be made at the most senior level, decisions are likely to be too slow and batch size will inevitably increase. Devolving authority to the team who take responsibility for understanding the business objective and costs of the project, enables them to continue with small batches. If a company values flexibility, then enabling its teams to work in this way becomes a priority.

## **Activity 4:** What dependencies influence your batch sizes?

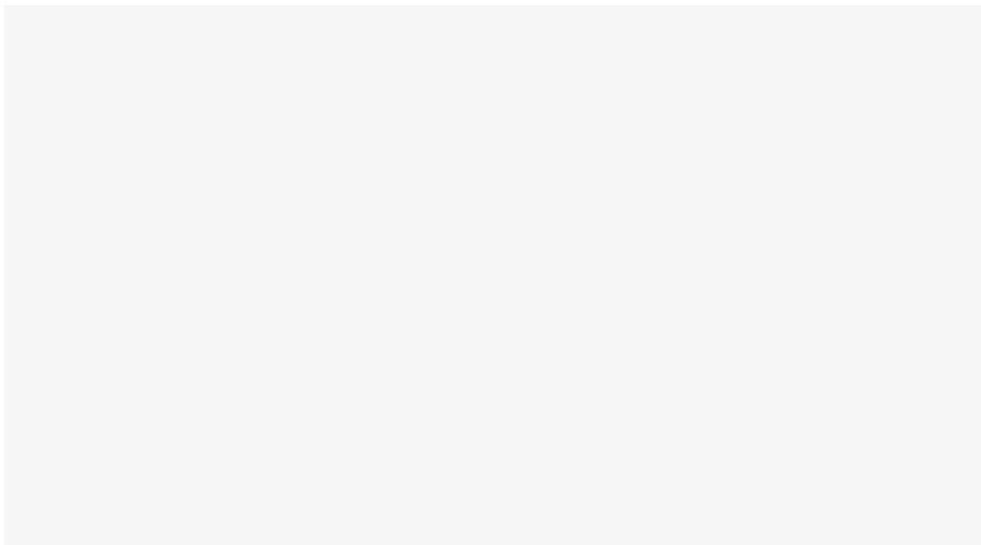
As you have just read, large batch sizes are due not only to the setup of development activities but often result from external dependencies in the organisation.

This activity should take about 10 minutes.

Look back at Activity 2 where you estimated your batch sizes. Examine more closely inputs to the development processes.

What dependencies exist in your organisation that have an influence on batch sizes ?

Once you have identified the dependencies, pick one and consider what changes would need to be implemented in order to allow smaller batches to be processed.



---

## Scope

Some projects seem to attract extra features. Perhaps we want to make the project look exciting, so we put in all the fancy ideas we can come up with. Maybe we want to make a launch seem particularly impressive and so we want to be sure it has more features than the last version.

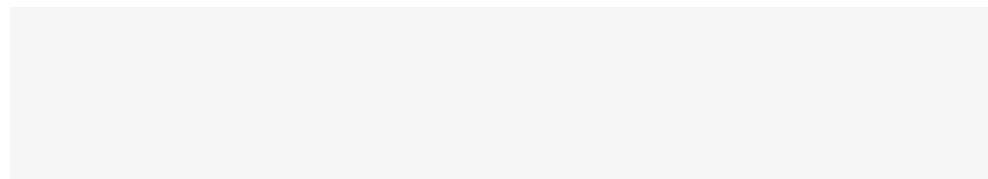
When we pack extra features and innovation into a single project we are increasing the batch size. Indeed, any kind of 'scope creep' increases the batch size. Normally the 'extras' are unnecessary to the project's key aim. They simply get in the way of an early launch, which would enable feedback. A few new features are typically enough to make a compelling story and generate interest – any more is just as likely to confuse as impress. Secondary features can always be added later.

---

### Activity 5: The appeal of big

This activity should take no more than 5 minutes.

Pick a CV. It could be yours, or you may ask a colleague for one. Perhaps you are doing some recruitment and have a few at hand. Read through the document and highlight all mentions that refer to large size, whether of project, scope, budget or company. How many did you find?



#### Commentary:

How many of these large items were actual outcomes, for example profit, and how many mentioned the size just for effect, for example people or cost?

We want to portray our experience in the best light possible. Large values, we consider, are impressive and will look good, so there is a natural instinct for our staff to want to work on big projects. However, big projects often mean large batch size with all the associated problems – something CVs rarely discuss! It might be interesting to see if the person whose CV you have looked at has examples of small, incremental successes to talk about in discussion.

---

So why do we continue to increase scope? The answer seems to be that big projects satisfy something inside us. Working on the transformation project which is going to change the way we do business forever has more cachet than making the way our orders are processed a tiny bit faster. Yet the latter aim may be part of a transformation, and it may well be best to launch this first, before going on to improve something else.

Thinking like this runs throughout the organisation. CTOs or CEOs are often keen to create a single big project that they can point to as their 'legacy'. Yet such projects are more likely to fail – and the embarrassment of failure, let alone the economic cost, is severe.

---

**CASE STUDY:****Ford sets off to conquer Everest**

Figure 14. Mount Everest

In 2004, Ford launched a new supply-chain system based on Oracle software – called 'Everest'. Perhaps naming the project after the world's highest mountain on which numerous climbers have met their deaths should have been a warning. After years of development and huge investment, the results were depressing – angry suppliers, failure of integration and a decision to revert to the legacy system,

which the massive new 'Everest' had been intended to replace. The company said only that it planned to salvage some of the new capability and continue using the legacy system.

If the fallback plan of the old system plus new capabilities was truly workable, then it suggests that the initial grandiose scope of Everest was a mistake. Far better to begin by adding features and untangling the legacy architecture. A system that results in frustrated suppliers suggests a very inefficient feedback cycle that failed to validate assumptions – small batches with regular feedback would have avoided this. Finally, it also appears that stage gates weren't working as they should – the role of these is to kill non-performing projects, not allow them to roll forwards consuming more and more resources.

---

## Funding and stage gates

Traditional stage gates, as we discussed earlier, tend to lead to large batches, and a resulting slow cycle time. According to the Product Development and Management Association, the percentage of companies in North America using a stage gate system rose from 44% in 1995 to 73% in 2005. Ironically, the process was invented in an attempt to reduce risk and focus resources onto the right projects, but instead the process makes delays more likely and so increases risk.

Imagine the stage gate at the beginning of the process – this is normally where resources (both money and people) are assigned to a project. In order to justify spending a large amount of money, or taking people off other work, the company quite properly wants to see a projected business plan, possible risks, perhaps some research from customers on how much they want the product. Of course in order to provide this, the project has already had a large amount of work done on it, some of which may have already secretly bypassed the official stage gate (money may have been spent on research or prototyping). Because the project team wants to make sure they don't have to repeat the whole process in a few months to ask for more money, they throw in everything they might possibly want – leading to a massive batch size.

At every stage gate, an approval is required before the project can progress – at each of these points the batch size is thus at its maximum. This means that faster bits of the project have to stop and wait for slower bits which ensures the project will move at the pace of the slowest element. And that's not counting the time the project will spend in a queue waiting for the stage gate board to convene (are they held as and when required, or once a month?) and deliberate.

We do not believe that small batches should bypass governance altogether. Simply that the stage gate process as it is usually implemented, does not serve its intended purpose. Stage gates need to be able to cope with small batches – which have their own specific needs and up-front investment. For example, devolving authority to the team for specific amounts of expenditure or decisions will enable the process to move faster. Today, a good example of how this can work is found with venture capitalists.

At the start of a business, when the ideas have a high chance of failure, venture capitalists invest a small amount to allow a business to test out concepts. With some hopeful early results, a further sum will be invested – this allows the business to 'prove' their business case on one or two of the most promising ideas. A much larger sum is required to actually develop the idea, and finally the largest amount of investment comes with launching the product. Risk is reduced because the money invested is split into smaller chunks and progress is reviewed regularly.

## Planning

Part of our obsession with on time delivery is that we begin every project with a detailed plan. This lays out the milestones throughout the project, decides on resource needed at each stage and allocates it accordingly. Of course, to do this with any rigour means knowing the full specification of the project, with analysis and detail for each requirement. And that precipitates the problems discussed above.

Instead a project plan should sketch in the big picture, keeping a short horizon for detail – perhaps 30 days or so. This allows the project to stay flexible, while still keeping an acceptable level of control over the long term. Apart from anything else, it saves all those wasted hours spent by a conscientious project manager creating a plan that turns out to be utterly redundant within a few weeks.

### 4.3. So why do we still have large batches?

Do the benefits seem obvious?

They may do, but the fact is that companies continue to implement large batch sizes, because a powerful rule of thumb tells them that making a change is risky. Releases are an excellent example. Every time you make a release, there are bound to be new bugs, new problems – a chance of failure. This is scary. The result is that companies decide to do fewer releases. But reducing the number of releases rarely reduces the amount of change. It just means that our single release is much bigger, which actually increases the number of things likely to go wrong and the chance of failure. Instead of realising that a large batch size is at fault, most organisations assume that the process of release itself is the problem.

The result? They release even less frequently, making the batch size even larger...

Even if it sounds obvious, reducing your batch size is not necessarily easy. Examine your own working processes carefully, and you'll find there are likely to be some large batch sizes disguised as a means of reducing risk. Implementing smaller batches is likely to require some changes and adjustments within your organisation that may be difficult. But long term, you will reap the rewards. Companies that release software regularly tend to have fewer failures than those who release less frequently.

## Activity 6: It's party time!

This activity makes a good warm-up as part of a regular team meeting. It will take approximately 15 minutes.

You have decided to organise a party and need to create invitations for your guests. You've gathered some friends to help you out with this task.

**Objective:** Create 18 good quality invitations for your party as quickly as possible.

**Preparation:**

1. 36 pieces of paper (18 for each round), A5 size will be sufficient.
2. Coloured pens.
3. Stickers or a stamp.
4. A time keeping device.

**Batch:** A set of invitations processed at any one time.

**Batch Size:** The number of invitations in a Batch.

**Card Instructions:**

1. Fold a piece of paper in half.
2. Write "Party Invitation" on the front.
3. Inside the card draw a simple picture.
4. Attach a sticker or stamp the finished card.

Before you begin create a prototype of an invitation together and agree what a finished product should look like.

**Play:**

Each person selects one task from the card instructions to do. Tasks must be done sequentially in the order given.

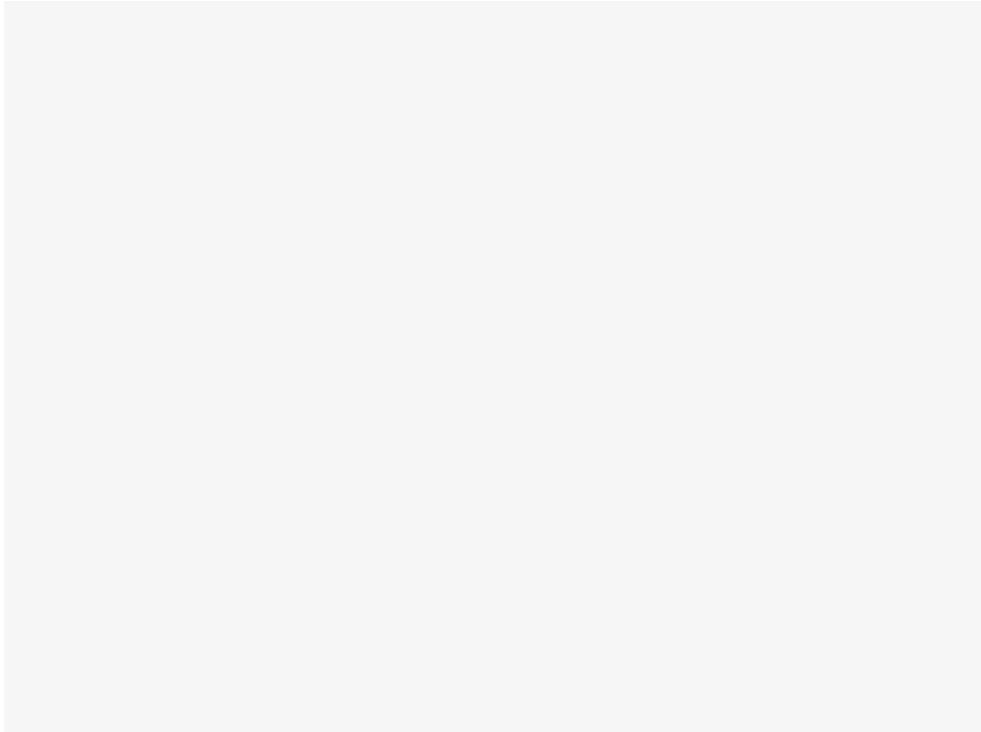
Start the first round in which every person should work with a Batch Size of six passing the invitations from one person to the next. Measure the time it took to complete all 18 invitations.

Create another set of 18 invitations in round two. Use two as your Batch Size this time.

**Rules:**

Only pass the work to the next person after the whole Batch is complete.

If there are any defects with the work up-stream return the whole Batch.

**Commentary:**

How quickly were you able to complete your invitations in Round 1? How did the time change in Round 2?

What do you think would be the effect if the Card Instructions were changed during a round?

Typically we find that changing requirements during an activity has far greater implications if the batch size is large. Hence in traditional software development we aim to lock down requirements before the work starts. Reducing the batch size makes it easier and cheaper to adapt work in progress.

---

# 5

## THE TROUBLE WITH I.T.

We've looked at some of the difficulties at making batches smaller, but also seen why overcoming them will produce benefits in speed, flexibility and reduced risk. So what stops us?

### 5.1. 'Ah yes, but THIS project is different'

Most people agree that releasing an increment every 3 to 6 months is less risky and adds more value than a huge release every 2 years. But when it comes to their particular project, they argue, change just isn't possible...

We have looked at some of the most common dependencies that need adjusting to enable smaller batches, but it is not an exhaustive list. Each project is different and has its own unique dependencies and blocks. These are often used to justify large batches.

---

#### The Emergn View

We once worked with an organisation that insisted frequent reviews were impossible. There was a very real constraint: the most senior figure who had overall responsibility for the project, simply did not have time for weekly meetings. Well then, we suggested, why not devolve authority to someone who did have time to attend frequent reviews. Equally impossible! they declared. The senior figure must be involved with every increment. The project was late and underperformed against expectation – a direct result, we believe, of the overly large batch size.

So our conclusion is that if you don't like late projects with a high risk of failure, then it's worth tackling the dependencies – however painful they are.

---

The evidence is that small batches work for almost all types of software development in several industry sectors. The Agile Scaling Model, lists the range of systems that currently benefit from such an approach:

"web-based applications, mobile applications, fat-client applications, business intelligence (BI) systems, embedded software, life-critical systems, and even mainframe applications... applied by a range of organizations, including financial companies, manufacturers, retailers, online/e-commerce companies, healthcare organizations, and government agencies.

Some organizations... are applying agile techniques on large project teams - hundreds of people and on distributed teams, in regulatory environments, in legacy environments, and in high-complexity environments."





## CONCLUSION

Small batches are a powerful tool to assist faster delivery in IT – by reducing queues they allow us to eliminate waste without changing either the total amount of work done or the capacity required to service the work.

Enabling effective use of batch sizes is not always straight-forward, but the cost of not doing so is high. For companies with a need for flexibility this is a key step to implement. Remember the new rule of thumb: your batch sizes should probably be smaller.

Now that you have completed this session, you will be able to:

**Understand what a 'batch' is in the context of software development**

- To discover the size of batches worked on in your organisation

Appreciate why small batches lead to fewer queues and faster cycle times

- The power of small batches and regular arrival of work to optimise flow
- Smaller batches reduce queues and hence delays

Recognise that small batches also decrease risk through faster feedback, while increasing urgency

- Important learning can often come from a small release
- Regular feedback prevents wrong decisions
- Immediacy created by small batches engenders urgency and motivation for those working on the batch

Understand the relationship between batch size and transaction cost

- As batch size decreases transaction costs will eventually rise
- Savings or extra earnings created by faster development can be reinvested to reduce transaction costs

Identify ways to reduce batch size by time, input or size

- It's not always simple or obvious how to divide batches up
- Reducing input to a batch (limiting features or scope) is the simplest and cheapest way of reducing batch size

Identify the likely catalysts of large batch size within software development processes

- Funding, stage-gating and approval processes all tend to create large batch sizes
- Up-front specification, detailed schedules and large releases are also likely culprits

## BIBLIOGRAPHY

- Anderson, D.**, 2003. Agile Management For Software Engineering: Applying The Theory Of Constraints For Business Results. Prentice Hall.
- Application Development Trends**, 2004. Oops! Ford And Oracle Mega-Software Project Crumbles. Available at: <<http://adtmag.com/Articles/2004/11/01/Oops-Ford-and-Oracle-megasoftware-project-crumbles.aspx>>. [Accessed 9 January 2012].
- Cramm, S.**, 2010. 8 Things We Hate About IT: How to Move Beyond the Frustrations to Form a New Partnership with IT. Harvard Business School Press.
- IBM**, 2009. The Agile Scaling Model (ASM): Adapting Agile Methods for Complex Environments. [online] Available at: <<ftp://ftp.software.ibm.com/common/ssi/sa/wh/n/raw14204usen/Raw14204USEN.PDF>>. [Accessed 9 January 2012].
- Larman, C.**, 2003. Agile & Iterative Development: A Manager's Guide. Addison Wesley.
- Larman, C.**, Vodde, B., 2008. Scaling Lean & Agile Development: Thinking And Organisational Tools For Large Scale Scrum. Addison-Wesley.
- Leffingwell, D.**, Widrig, D., 2010. Agile Software Requirements: Lean Requirements Practices for Teams, Programs, and the Enterprise. Addison Wesley.
- Reinertsen, D.**, 2009. The Principles of Product Development Flow: Second Generation Lean Product Development. Celeritas.
- Schwaber, K.**, 2011. Happy Holidays. Ken Schwaber's Blog: Telling It Like It Is. [blog] 25 December, Available at: <<http://kenschwaber.wordpress.com/2011/12/25/255/>>. [Accessed 12 January 2012].
- Shalloway, A.**, Beaver, G., Trott, J., 2009. Lean-Agile Software Development: Achieving Enterprise Agility. Addison-Wesley.
- Shingo, S.**, 1996. Quick Changeover for Operators: The SMED System (Shopfloor Series). Kindle Edition. Productivity Press. (Kindle).
- Stage-Gate International**, 2007. Product Innovation Guru Dr. Robert G. Cooper and Nine Leading Companies Address Business Executives at the 1st Annual Stage-Gate® Leadership Summit. [press release], 13 March 2007, Available at: <[http://www.stage-gate.com/news\\_031307.php](http://www.stage-gate.com/news_031307.php)> [Accessed 10 January 2012].

**NOTES**

---

**NOTES**

---

**NOTES**

---

**NOTES**

---





[valueflowquality.com](http://valueflowquality.com)

[emergn.com](http://emergn.com)