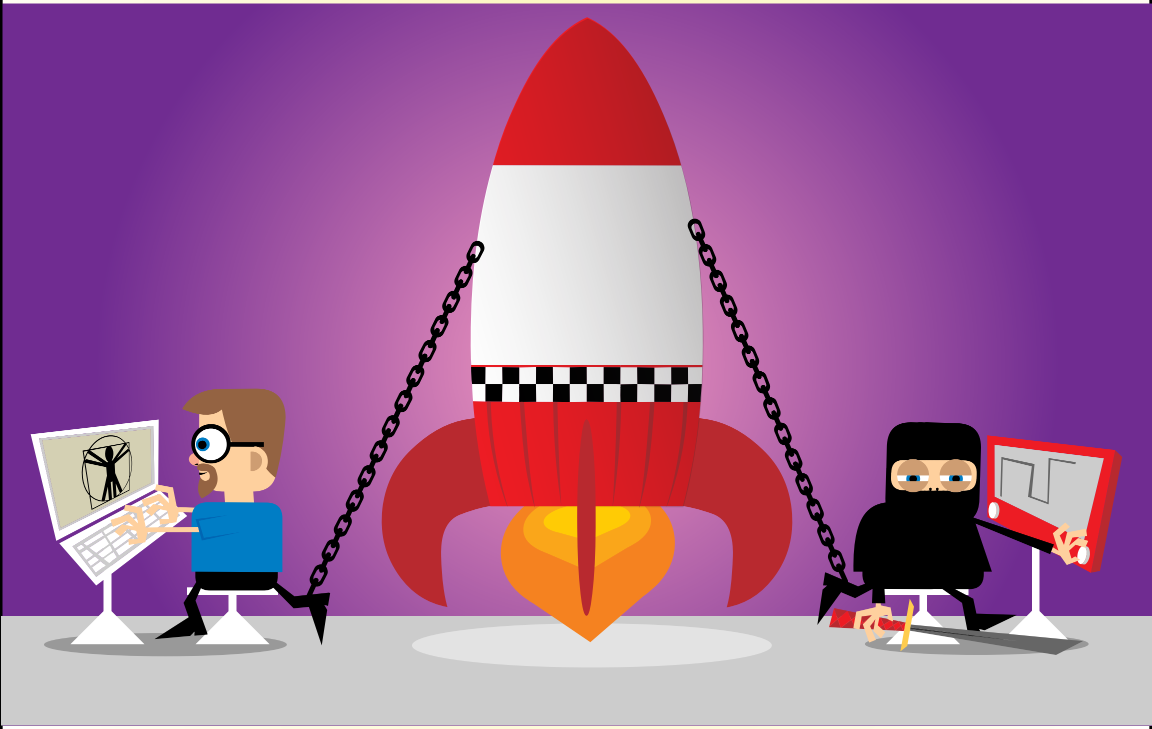
ROCKET FUELED PROCESS

ADVANCED LEAN PRODUCT DEVELOPMENT FOR SOFTWARE STARTUPS

BY WILLIAM BELK



"Software businesses are challenging. The startup environment is completely unpredictable. Teams need support structures to combat complexity and risk."

v 1.4

Dedicated to my father, my uncles and Matthew Otto,

my first software mentor, who introduced me to lean

and agile software principles.

Who is this book for?

While the contents of this book focus on software product development, its lean principles can apply to any type of product development. As software product developers we often intersect with the distribution of physical products. We also find ourselves exposed the entrepreneurial side of the industry through fundraising, reporting and testing new product concepts. Lean transcends all products, disciplines and services. As such, this book is for all builders, entrepreneurs and designers.

All illustrations by Dave Savage | www.savagemonsters.com

Table of Contents

FOREWORD 8

CHAPTER 1: PRODUCT 101 11

The Beauty of Deconstruction 12

User Centered Design 13

The Declaration 15

CHAPTER 2: REALITY AND CONSTRAINTS 16

Traditional Models of Product Development 17

What is the Reality of Building Software? 18

Mastery & Unrealistic Behavior 19

Prioritize The Real Value We Are Creating: ROI Reality 19

The Beauty of Constraints 21

CHAPTER 3: INTRO TO LEAN 23

What is Lean? 24

Lean Startups and the Democratization of Learning 25

Concept to Iteration 26

CHAPTER 4: TOTAL COST OF OWNERSHIP 29

Understanding Total Cost of Ownership - Lifecycle Economics 30

Understanding The Impact of Debt 33

Debt In Code 33

Debt In Culture 34

Debt In Process 36

Debt In People 36

CHAPTER 5: BATCH SIZE & WORK IN PROCESS (WIP) 39

Understanding Batch Sizes and Risk 40

Implementing Work in Process Constraints 42

Eliminating Unnecessary Work In Process (WIP) 42
Synchronicity Is Essential 43
Synchronize Small Batches Around Common Goals 44

CHAPTER 6: TOOLS & METHODOLOGIES 45

Tools are just artifacts of our foundation 46
Overview of some popular development methodologies for startups 46
Agile 47
Scrum 47
Step 1: Planning Meeting 49
Index Cards 49
Step 2: The Sprint, also called the Cycle or Work Iteration 49
Step 3: The Daily Standup 50
Step 4: Retrospective 51
Kanban 52

CHAPTER 7: THE SIMPLE TEAM PRINCIPLE 53

The Simple Team Principle 54
1. Simple Scope Agreement 55
2. Simple Build Agreement 55
3. Simple Validation Agreement 56
We Have Completed One Iteration 57

CHAPTER 8: STRUCTURING TEAMS FOR SUCCESS 58

Building Ethical Systems 59
Team Structure 60

CHAPTER 9: SKETCHES, WHITEBOARDS & WIREFRAMES 62

Do It Like Picasso 64

CHAPTER 10: FINANCIAL MODELING 66

CHAPTER 11: KEY METRICS & SUCCESS MEASUREMENT 70

Metrics and Validation Analysis 71

Team Measurement 72

Micro Experiment Testing & Optimization 73

CHAPTER 12: WHAT MAKES A GREAT DEV TEAM 74

Product Managers 75

Experienced Engineers 76

Systems Engineers 78

Hiring Personalities 79

CHAPTER 13: GO MAKE A CUSTOMER HAPPY, TOMORROW 80

CHAPTER 14: Random Musings 82

Good companies are always in Beta 83

Advice for Small Startup Teams 83

Advice for Small Teams in Large Organizations 83

Advice for Product Managers 84

Advice for Engineers 84

About the Author 85

FOREWORD

Software businesses are challenging. The startup environment is completely unpredictable. Teams need support structures to combat complexity and risk.

With the world as its stage, the promise and limitless scale of the Internet has arrived in a major way. Our time online only increases with connected devices. IoT is just emerging. We trade online. We date online. We work online. Our workout data is online. Our cars are now online. Even our ten year old nephew has an online persona.

With more than a decade of rapid growth in the software-as-a-service (SaaS) and platform-as-a-service (PaaS) IT segments, the cost of experimenting with software products continues to decline. Companies like Amazon, Heroku, GitHub, Google, Facebook, SalesForce and TravisCI have fostered untold innovation by offering their products as a service to companies who pay for usage or contribute to their platforms with value-added products.

In addition to the ‘platform as a service’ model shown by Amazon and others, innovation is being fueled by a maturing machine of company production. Angel investors are frothing. Seed funds are frothing. VCs are frothing. The IPO market is always cycling. ‘Getting funded’ has never been easier; there has been a wholesale democratization of creating Internet startups.

All of this innovation creates a great deal of excitement. While we exist in an industry where it is surprisingly difficult to become profitable, there are few other places where a twenty person team can generate $1B in perceived value in 24 months. The mythical lore of Internet founder millionaires has been romanticized from blogs to movies.

However, the road to Internet stardom is not paved with gold. The start-up lifestyle is one filled with risk, stress, and in many cases, poor quality of life. If we are fair to assume that only 2% of startups achieve a positive liquidation event, there must be powerful realities that dictate how we should, or should not build. One of the largest factors in product development is complexity. Complexity in software business systems is created by a dynamic and organic environment where innovation fuels change, capital fuels more innovation, and code must be able to change as quickly as the business, often pivoting overnight toward a desired path.

Software businesses are challenging. The startup environment is completely unpredictable. Teams need support structures to combat complexity and risk. Without them, failure is almost certain. Successful software builders and entrepreneurs do not ‘get lucky.’ They learn their way to success. They learn how to influence their outcome. They learn how to build sustainable businesses, people-centric systems, and efficient processes.

Lean product development becomes ever more important as the speed of business becomes realtime. Applications and services are faster, information is faster, the financing channels are faster, thus our development processes must be faster without sacrificing quality. One can have the most innovative ideas in the world, but if a development process cannot support the production of those ideas, the product or business will not achieve its full potential.

CHAPTER 1: PRODUCT 101

It's happening all around us. People are having serendipitous episodes, and it all started with a product designer, somewhere, tucked away, cranking on something special.

Product designers exist to solve problems, to reshape the world, to invent the what and why, noodle about the how, and shepherd new ideas and experiences into the world. They share the same drive as traditional engineers, as scientists, as mathematicians, as physicists.

The Beauty of Deconstruction

Many great products have been born from the deconstructive pursuit. Someone sitting on a stool, leaning back, trying to understand and rethink the big picture. Twelve years ago art theory changed my life. A group of my peers critiqued things that I created painstakingly over hours, days and weeks. For the first time in my life, I had to defend to a group of people why I created something. Up until that point, I just built for the sake of building. For some months, maybe years, this deconstructive ritual put my mind into a fragile state. Coming to terms with the fact that I had almost no real understanding of why I was building things, or more importantly with the outward absurdity of some of my logic, was incredibly difficult.

Trying to peel back the layers of something and target the essence of intent can have profound results. This began to make itself evident to me in artworks like Marcel Duchamp's Ready Mades, David Hammon's selling snowballs in Harlem, Carsten Höller's Light Wall, Dan Flavin, James Turrell and Steve Irwin's light works.

Just as in art, deconstruction in software process has proven powerful, if not revolutionary.

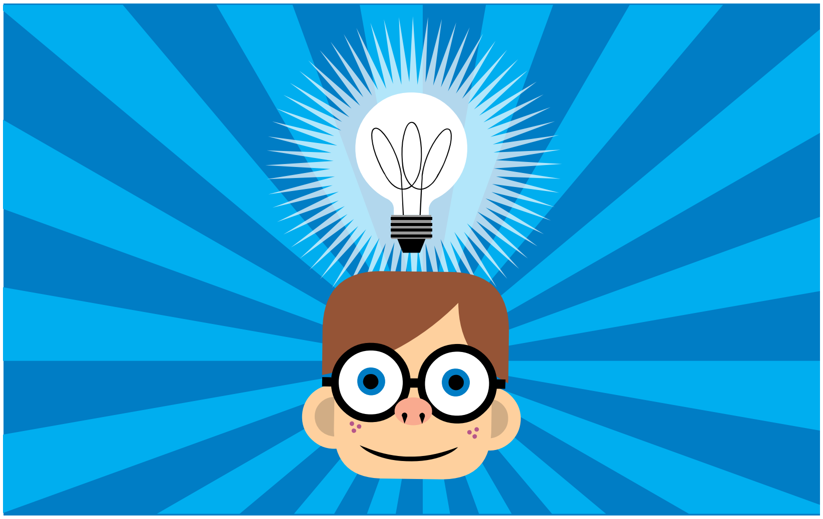
User Centered Design

One of the most compelling aspects of concepting and designing new software features is the blank canvas. One gets the opportunity to test their ideas about what software should be tomorrow, often with little responsibility to yesterday's assumptions. The blank canvas comes with a great deal of responsibility. As product designer, who do I choose to be accountable to?

Product development is so obviously about the end user. Every product designer understands this, right? Great product designers are able to shield influences from their personal and professional lives, the good and the bad, and focus on how to provide end users with the best experience. Quite honestly, the software world is littered with inexperienced product managers, UI designers, software engineers and founders who make decisions based on strange emotions, social perception, organizational pressures, and sometimes just because they are stubborn asses. It is very easy to avoid this behavior.

Empathy. The most successful product designers embody it. They empathize with the end user. There is a serious problem that needs to be solved. End users don't care what programming languages we use, how many provisional patents we've filed for, how many Stanford MBAs we have on our team, what our post-money valuation was, who our advisors are, or how many instances we're running on Amazon. End users want serendipity. Someone behind the veil really understands them.

Look around at all the happy people. Her first iPhone. My relentless Instagram addiction. Her AirBnB travels. His personalized Nike IDs that say "Suck It, USC." My sister's Toyota Prius. Lady Gaga's Little Monsters. Yuppies and their Vita-Mix blenders. The tourists at a Los Angeles beach stuffing their face with In-N-Out burgers, animal style. It's happening all around us. People are having serendipitous episodes, and it all started with a product designer, somewhere, tucked away, cranking on something special.



Simplicity. The most successful product designers embrace it. Simple products are easy to use, easy to engage with, and easy to place in one's life. Simple products cost less to maintain. Simple products can be brought to market quickly.

Look at Craigslist. A beacon of simplicity and still an example some people love to hate. Look at Subaru. Half of their vehicles use the same chassis. Most of their vehicles can share the following essential components: engines, transmissions, brakes, seats. Look at Instagram, one of the fastest growing mobile applications in history. It all started with low resolution imagery, no zoom, no swiping, nothing extra. Instagram is all about sharing a moment and social reciprocation. Take a picture, add some contrast and special sauce, share it. Double tap to like. That's it. So simple, it's the essence of a feedback loop.

The Declaration

As product designers, we must be able to communicate with others who will help us build our products. These people can be investors, engineers, marketing folks, business partners.

Sometimes we are working at the feature level. For example, building a review system for e-commerce products, or a REST API. Sometimes we are conceiving of a larger project like an entire startup.

When working at the startup level, we should be able to explain the essence of a product in one sentence. In so doing, we set the tone for all development to come. If it takes more than thirty seconds for someone to truly appreciate your value proposition, you're in serious trouble.

Here are few great examples of product declarations from the web and beyond:

- **"Google perks at CostCo prices." - BetterWorks**
- **"Expenses don't have to suck." - Expensify**
- **"When it absolutely, positively has to be there overnight." - FedEx**
- **"The world's online marketplace." - Ebay**
- **"Organize and share the things you love." - Pinterest**
- **"All your travel plans in one spot." - TripIt**

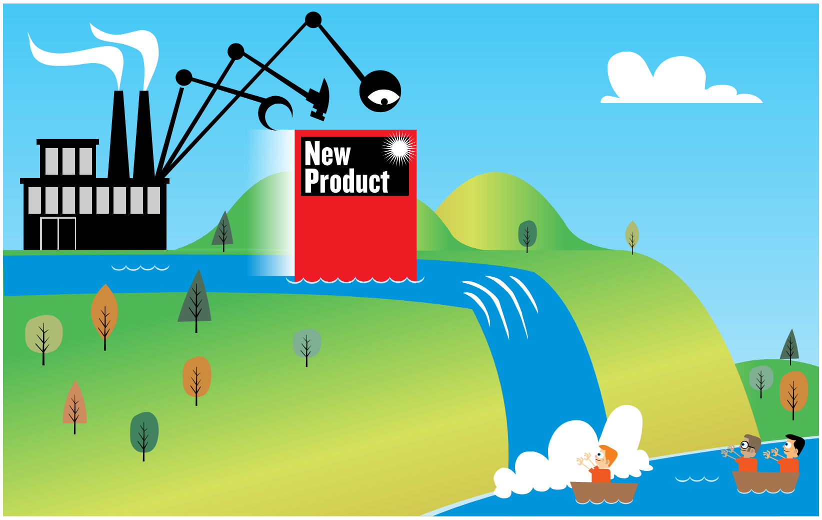
CHAPTER 2: REALITY AND CONSTRAINTS

The young developer tends to underestimate complexity and under-appreciate the true mastery required to design and maintain software at the elite level.

Traditional Models of Product Development

Traditional models of software product development view the world as static. This view often separates the ‘knowers’ from the ‘builders.’ Strategy and Product people are paid to plan the product. Engineers are paid to build it. This type of world view most often generates the Waterfall process of product development.

The term Waterfall Development describes a process where a monolithic product is shipped to the customer on a specific day and time, with all assumptions that influence the product remaining shielded from customer input during the development period.



The Waterfall process needs certain things in order to retain its authority. The Waterfall process needs blind assumptions. After all, if we have a process that does not need streaming customer feedback, then we do not need to accommodate change. If we cannot accommodate change, then we must be correct in order to succeed.

The Waterfall process needs documentation, and lots of it. If we're not worried about learning along the way, then we don't have to worry about our documentation changing. As such, we will probably want to front load all of our work so we can hand over a 120-page PRD (product requirements document) to our engineers and continue working on our golf game while we wait for them to finish.

Once upon a time, software was shipped on disk and the Waterfall process was king. There was no such thing as streaming customer feedback. Oh, how times have changed...

What is the Reality of Building Software?

Traditional models of software development have a warped, idealistic foundation. Successful software systems are organic, dynamic. Building software is an empirical process. Every next piece of information reshapes our reality. No two companies are the same. No two codebases are the same. Furthermore, software systems are built by people. No two founders are the same. No two teams are the same. Something could happen on a Monday that requires an entirely new direction by Friday. One of our employees can quit. One of our service providers can go out of business. One of our competitors could beat us to market by two months.

Because software business systems are so dynamic, our development processes must accept change as a constant. We must build frameworks for learning, thinking, and workflow control that better our chances of success.

Mastery & Unrealistic Behavior

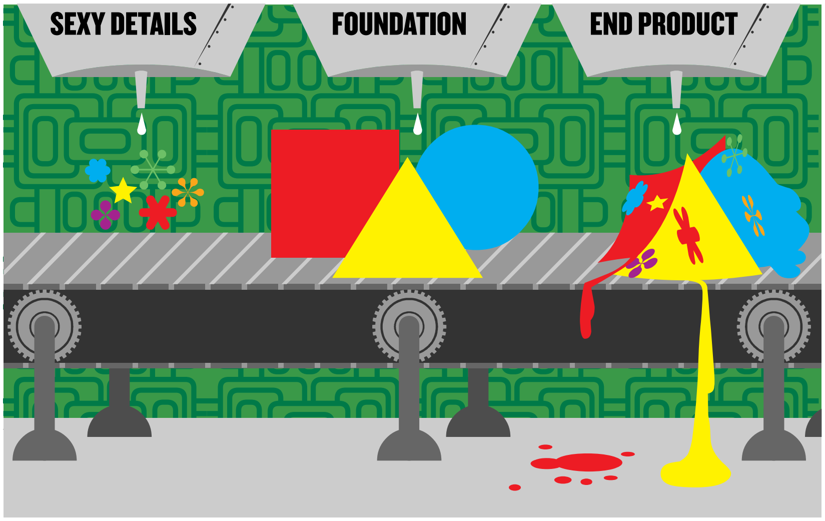
It is often said that in general pursuit, mastery comes after 10 years or 10,000 hours of practice. This is certainly no different in the software startup world. To be sure, some young entrepreneurs burst onto the scene with perfect timing, doted on by investors and suitors alike, and enjoy an early exit. However, for the other 99% of product developers, it's a hard road. It takes years to develop the personal relationships and analytical tools necessary for success. The mythically optimistic approach imbued by many accelerators and young teams borders on ridiculous.

Until a person has ventured into the depths of mission-critical systems, it is impossible to respect and appreciate the nuanced complexity of building and maintaining proprietary software systems. Quite so, young engineers and inventive types, with all their innocence, magical problem solving, beautiful creativity and mystical curiosity, cannot be trusted. The young developer tends to underestimate complexity and under-appreciate the true mastery required to design and maintain software at the elite level. The young developer needs thinking frameworks and tools to help navigate the risky waters.

Prioritize The Real Value We Are Creating: ROI Reality

Distraction is often the order for young product developers and entrepreneurs. Most of the people reading this book have at some time been perplexed seeing someone obsess over something completely unrelated to a product's core value. Wasted cycles go by, confused employees, advisors and investors see little or no return on investment. Most importantly, end customers vote with their time and use other products that better solve their problems.

This 'cart before the horse' distraction is common in the incubation stages, where product developers sometimes fixate and obsess about future scenarios that are entirely gated by the fundamentals. If we cannot take a credit card online, and send an order to our warehouse, no amount of social-K-factor-sharing-economy-genius is going to help us generate revenue. If we have an ad-based revenue promise, worrying about selling ads to advertisers should surely come after we figure out how and why we are going to build an amazing destination to get eyeballs, combined with a deep understanding of how much each visitor will cost us.



Young developers also tend to measure their future feature set against the monolithic market leaders like Google, Amazon and Facebook, overlooking that these giants have had teams of 50+ engineers working for 5-10 years developing and refining features. No amount of optimism can replace time and man-hours. Keep it simple.

We must distill and master the core of our business first. Arrive to market before our competitors. Compromise and complete the most important, smallest increment of value. Then use our new perspective to solve the next problem. Often our priorities will change dramatically over time. What seemed vital four weeks ago may be of trivial importance tomorrow.

The Beauty of Constraints

Some of the most creative and beautiful examples of product development and engineering are born from situational constraints, or boundaries that must be respected.

Americans traveling in Tokyo have experienced the crisp efficiency of Japanese spaces. Europeans traveling in Indonesia learn to love the banana leaf food pouch that is made to eat without utensils. Javascript developers show their skills by building entire applications that occupy less than 1K of space. Turntablists create complex compositions using only records and a mixer. Conceptual artists once reinvented the painted canvas through minimalism.

Successful entrepreneurs and product developers thrive within a system of constraints. At the inception of each new product they use steadfast discipline to define the constraints within which they must operate for the economic benefit of the business.

Any team who has relied on external service providers is familiar with compromise and constraints. Successfully operating within constraints can afford them competitive advantages like rapid time to market, reduced risk profiles, lighter personnel needs, faster learnings, and so on.

To use an example that surely arises in any modern dev shop, a great mobile product developer understands the cost of application updates and operating system compliance, the places where they absolutely must have native interfaces, the places where they can use a webview to minimize the cost of change and share components across platforms.

Further examples of software development constraints are testing and code practice requirements. Faced with the gross cost of poor code quality, great engineering leaders first provide their teams with the automated tools to write, enforce, test and deploy high quality code before writing even a single line of application logic.

Time can also be a powerful constraint. Software hackathons are a great example of this. Teams agree to design, build and demonstrate a working prototype in 12-24 hours. Watch a winning hackathon team in action, and you will witness a world of disciplined compromise for the sake of the end product.

In the end, the business world is one giant hackathon. It seems each year the world moves faster and faster, and to the fast go the spoils. For the fast and learned product developer, efficiently validating her assumptions with minimal risk, there is no equal. How do we become her? How do we develop thinking and learning frameworks that allow us to manage time, resources and risk? How do we deconstruct our practice and improve our chances for success in this crazy industry?

The answer is Lean.

CHAPTER 3: INTRO TO LEAN

With a lean process we try to rapidly and predictably produce the smallest synchronized batches of demonstrable customer value.

What is Lean?

Like most things in our complex software world, the definition of Lean is simple, yet its tool set is ever changing, and its workflow has yet to be completely solidified. Quite simply, lean is about continually improving a development process while removing waste from the system. Lean is a state of being, a practice, a deconstructive pursuit that encourages reality and learning over assumptions.

The more information we have about something, the better prepared we are to respond to its needs. If we have information coming in small batches, we will be able to respond to that information much more quickly. The faster we build and ship software, the faster we learn. The more we learn, the more we validate our hypotheses and assumptions.

Conversely, the more waste we generate, the slower we become. We inhibit our ability to execute, to respond to market changes. Because we all work for businesses that pay salaries, our bottom line is always financial. If the factors of technology, people, product, structure or processes prevent us from responding to change and innovating for our financial benefit, or for the benefit of our customers, those factors must be deconstructed and improved in order to build a more profitable and sustainable business.

If you're reading this book, you likely already know of the origins of lean coming from places like the Toyota Production System in Japan. The Internet is filled with detailed information about the historical origins of lean. As such, I will not bore you here.

Lean Startups and the Democratization of Learning

We have seen an explosion in the 'lean startup' community since 2009, conservatively. These ideas are not new. Their foundation brought us process methodologies like Agile, Extreme Programming, Scrum and Kanban, automation frameworks like Chef and Rightscale, testing frameworks like JUnit and Rspec, continuous integration servers like Cruise Control and Jenkins, tracking tools like Jira and Pivotal Tracker, companies like Optimizely and Kontagent.

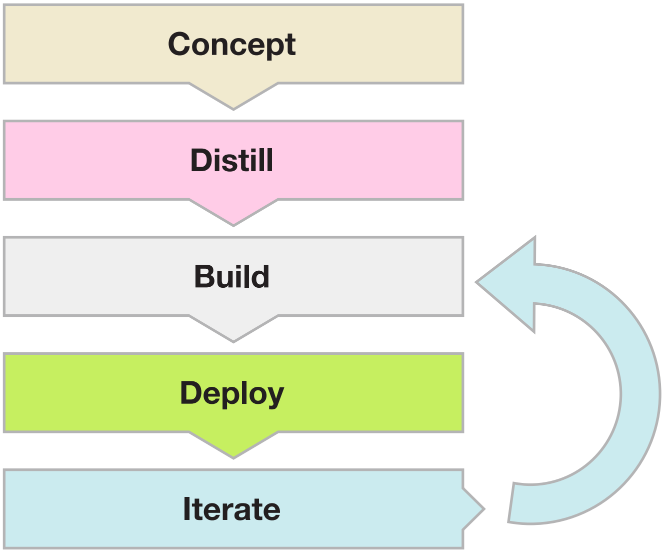
Steve Blank wrote some books. Eric Ries wrote a book. Donald Reinertsen wrote some books, one of which I consider to be the holy reference manual of product development economics called *The Principles of Product Development Flow*.

Most accelerator programs promote 'lean' startups. Contemporary developers and mentors associate being Lean with the ability to learn and act quickly, while keeping transparency close to 100 percent. In short, the lean startup methodology has become the thinking framework and metrics dashboard for learning about your software product. In 2007, we did not have this movement. It has created a framework and community that is democratizing learning for startups. Great stuff to be sure. Combine this democratization of learning with an unprecedented number of startup accelerators and angel investors, and our greater community has democratized having a startup altogether.

However, all of these frameworks have not democratized real innovation and authenticity. They have not democratized building products that the world really needs. It's easy to get caught up in the hype, excited by new information and tools. Before you jump in with both feet to the flames, make sure you're doing something authentic. Strive to be cool. Strive to deconstruct the world. Don't just create a service or commerce startup because you can. Don't just launch a product that lacks your emotional investment and listen to your random customers 100 percent; you are bound to offer the world yet another spiritless company. If your goal is to generate pure cash, startups are probably not your best bet. However, when you work with great people in such a fluid industry, few places can give you the excitement and pace of an Internet startup.

Concept to Iteration

Ultimately as software builders, our goal is to build and release working software in the shortest possible time cycle. We want to make the following funnel as efficient as possible, at the same time eliminating any waste that drags down our business:



With each idea or concept, we must distill. Any excess or 'nice to haves' must be eliminated. It's irresponsible to pass this excess on to our team members. KEEP IT SIMPLE.

Once we have distilled our idea, we build it with as little economic investment as possible. When building, we also try to incur the least long-term technical debt and, where possible, design re-usable components. We then release or deploy our product to real customers, gather validation data, and iterate on our product as necessary.

With a lean process we try to rapidly and predictably produce the smallest synchronized batches of demonstrable customer value. Sometimes the customer is external to our organization, sometimes the customer is under the same roof.

Now let’s get into the details.

CHAPTER 4: TOTAL COST OF OWNERSHIP

Without a framework for reducing risk, our blind assumptions can cost an organization incredible amounts of time and money.

Understanding Total Cost of Ownership - Lifecycle Economics

On our quest for process improvement and waste elimination, our fundamental goal is to reduce the cost of ownership (i.e. increase the efficiency and output) of our people and technology systems. Lowering our cost of ownership over time means that we gain resources: time, personnel, and cash—three things that are beloved commodities at any fast-moving company.

Every decision we make has an economic cost. Because we are inherently trying maximize the profit of our product, we must focus on reducing the costs and aggregated risk associated with our decisions and system maintenance.

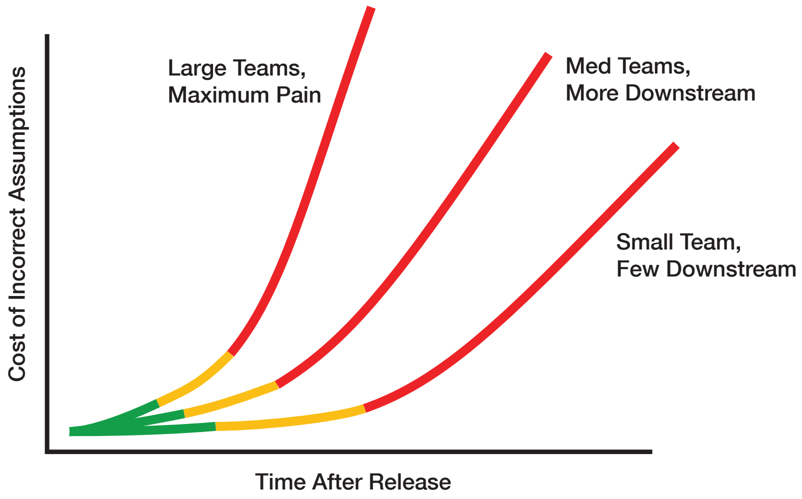
Too often team members lose sight of the big business picture, and instead focus on their personal needs as a product developer. They do not properly foresee how their decisions will affect downstream components. They make the decision that best serves their ego, creative pursuit or their sphere of influence. This must stop.

If our team is more than four people, when we present a wireframe, diagram or design, we are making a bold declaration on behalf of our team, all teams in our entire company, and our service providers. If, for example, we work at a large subscription e-commerce company, a single wireframe says:

- **Our backend team has sufficient capacity**
- **Our frontend team has sufficient capacity**
- **Our fulfillment team can support us**
- **Our content team can generate the required content to launch/release**
- **This feature will support our user acquisition team**

- **This feature will support our existing revenue streams, therefore our CFO will allow us to live for another few weeks**

The list above only takes into consideration the initial launch of a product. If we make incorrect assumptions, our cost of ownership over time is magnified by every part of our organization that it affects.



As an example of compounding economic cost, let us consider a pure technology product that is being built to support a team or business. Please consider the scenario below to illustrate the economic impact of poor process and poor decision making.

		Cost Per Month	Total Cost
Tech product released 60 days late	6 engineers	$12,000	$140,000
Marketing team cannot execute for 60 days	2 marketers	$6,000	$24,000
60 days of revenue lost		$50,000	$100,000
PRODUCT LAUNCHED			
Revenue projections were off by 50%		$25,000	$150,000
Too much buggy code at month 4, month 5 lost	Full team	$200,000	$200,000
			$614,000

We just cost ourselves over $500,000 in four months. Furthermore, we learned nothing during our development cycle because we did not work in small cycles and could not release any product. We are way behind our competitors in learnings.

Without a framework for reducing risk, our blind assumptions can cost us incredible amounts of time and money—not to mention the incalculable benefits of learnings gained through early time to market. Being conscious of the total cost of ownership of any piece of our system is the most important thing we can do for our process.

Every decision we make must be thought of with a goal of minimizing the economic impact over time. Are we accruing technical debt and bad code? Are we creating morale problems? Are we creating unsustainable production systems? Will we still be in business if we miss another deadline? Usually we get excited and assume that because we are working on something exciting, everything is great. However, building software is too risky to throw caution to the wind.

Understanding The Impact of Debt

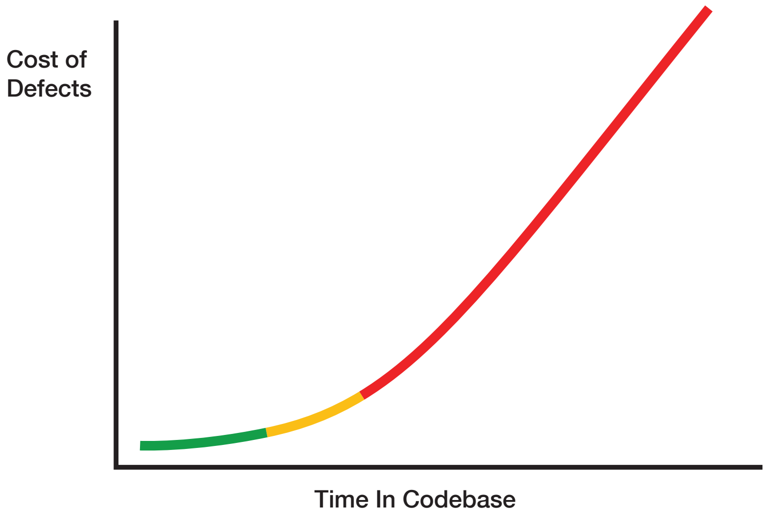
Technical debt is a common term used to describe the development of an unstable codebase or technology stack over time. However, debt is a term rarely used organizationally. Debt can accrue anywhere: in the code, in the culture, in the processes, in the people. Debt is waste. Eliminating waste through continual process improvement and efficiency is at the core of lean. Waste slows us down. Waste costs us money. Waste prohibits learning. Waste prevents a timely response to market conditions.

Debt In Code

Code defects/bugs are a form of debt. The cost of these defects can grow exponentially over time. The total cost is the:

- **Cost of the initial defect**
- **Multiplied by time**
- **Multiplied by number of components it ever touched**

Processes like Test Driven Development (TDD) use small batch sizes and a test-first approach to prevent code defects from entering the codebase before application code is even written. If some slip past, each defect is then corrected with an appropriate unit test to prevent that same problem from occurring again.



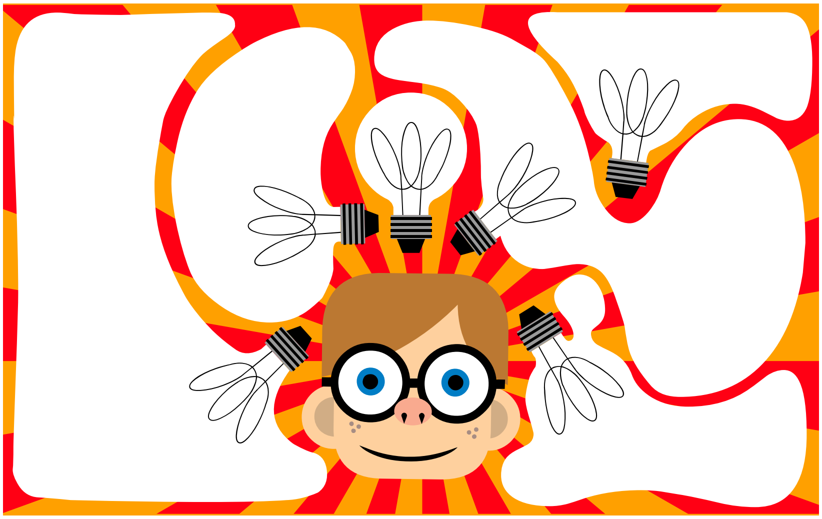
Technical code debt is a problem for many organizations. The start-up battlefield is littered with under-performers who lost the war against technical debt and were unable to achieve their full potential.

Debt In Culture

Company culture is one of the hardest things to understand. What I am sure of is that it originates from the top of an organization and filters down. While small teams within an organization can sometimes enjoy a measure of autonomy, surely no one is insulated from the culture set in motion by the founders and upper management. This can be a good or bad thing.

In my opinion, the worst culture that arises is the culture of time wasters, the culture of persistent thinkers. With the ranks filled with the non-operational likes of PhDs, MBAs, ex management consultants, life becomes about meetings, strategy, sign-offs, blueprints, PRDs, intellectual and political battles, posturing. It's tragic for the business, but more importantly tragic for the executors, the operators, the builders.

As Don Reinertsen once said in a Los Angeles lecture: "Ideas are like the perfect gas of physics, they can expand to fill any container."



We want a culture of actors, testers, builders. We rarely solve entirely new problems, particularly after we have identified a viable business model. We don't need philosophers, we need lean operators and executors. Surely a proper dose of creativity and innovation are essential for success, but I would argue that a room filled with a city's top consultants can generate a backlog of more innovative and quality ideas in one day than a team will ever have the capacity to act on.

Furthermore, without testing small hypotheses under real market conditions, ideas are just ideas. They are not written in stone. Just because they originate from a ‘genius’ doesn’t make them ‘genius’ until the market proves it.

Debt In Process

Debt in process generally arises from the traditional waterfall view of software development, as mentioned earlier. With its static and (wrongly assumed) predictable properties, batch sizes are large, release cycles are long, risk profiles are immense.

We see many artifacts from a life once led:

- **Excessive internal documentation**
- **Antiquated development methodologies**
- **Group meetings with poor agendas and tragic wastes of many people’s time**
- **20 person conference calls**
- **Minimal business process automation**
- **Excessive work in process, like prototyping too far in advance**

Processes that assume software development to be fluid, unpredictable and dynamic are far superior. Because our end product is software, teams must let go of the nonsense and start building. The real documentation of our ideas is reliable code, automation, and a product that has been validated by customers.

Debt In People

While debt in culture generally shows itself through an umbrella-like mist that starts to soak into the entire organization. Debt in individual people can be quite different, however with similar downstream effects. The most acute example is the single expert point of failure.

It is a sad situation to see, when early employees, who are the tenured product/engineering experts, become the single point of failure and the gate to progress. Heads of engineering sometimes suffer this fate. Because software systems are so complex, and time is so critical, tribal knowledge is often the norm. Without a structure for efficient and thorough knowledge transfer, this expert can find it very difficult to transfer processes and learnings. For the sake of the business, this person must successfully remove themselves from the equation and decentralize their knowledge. The best way is through automation. After all, there's nothing more efficient than showing someone in code the way a system is supposed to work. This expert person must also decentralize their knowledge to minimize opportunity cost -- they are much better served to be innovating and optimizing instead of maintaining.

Another form of debt in people comes from unjustified or overly ambitious empowerment. For example, how many times have we seen people destined for failure, simply because they are tasked with things that are outside of their sphere of expertise? Just because we want someone to own something and do a great job, does not mean they are fit for the task.

This is most often the case at the management level. Management is not an exact science, to be sure. However, we see many examples of managers lacking the technical understanding to efficiently communicate with, and earn the respect of their subjects. Just as in code, this kind of debt can have an compounding effect to all downstream components over the life of any misdirection.

CHAPTER 5: BATCH SIZE & WORK IN PROCESS (WIP)

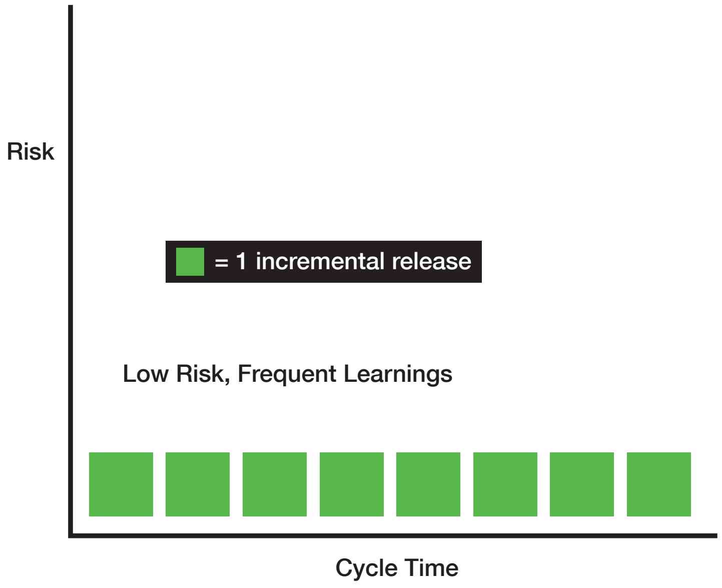
With the absence of a framework for implementing constraints on a development process, things naturally get out of sync.

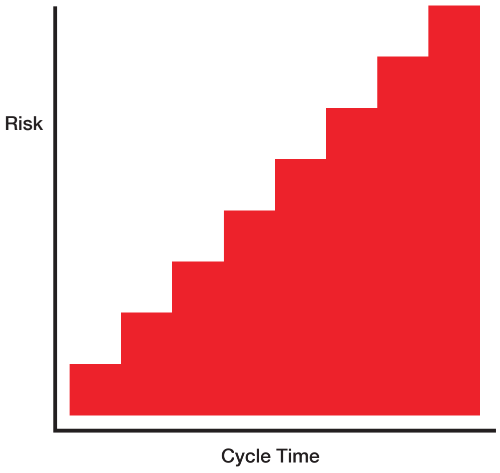
Understanding Batch Sizes and Risk

Batch size is the size, measured in work product, of one completed unit of work. Cycle time is the amount of time it takes to complete one batch of work. What we focus on with lean development is reducing batch sizes, thereby reducing cycle times, thus increasing potential learning points over time.

Batch size, risk and cycle time are all directly proportional. As any grows larger, so do the others. The amount of risk we create is equal to our batch size and cycle time, i.e. our total work in process. Risk shows itself in the form of total investment at risk (this could be labor, inventory, market opportunity cost, revenue, etc). As such, batch size is also directly correlated to the number of blind assumptions (i.e. untested hypotheses) that inform our product release.

As shown in the drawing below, the graph on top performs incremental releases over eight cycles. The graph on the bottom uses all eight cycles to perform one release. The process on the bottom generates 64 times more potential work at risk.





64 times as much Risk
8 times as much Cycle Time
0 Learnings before potential failure

Risk = Batch Size = Work in Process

Implementing Work in Process Constraints

Work In Process (WIP) is any ongoing work that has not been complet-ed. The largest and most identifiable component of WIP is batch size. As Donald Reinertsen states in The Principles of Product Development Flow, the most simple and powerful economic driver in a development process is a reduction in batch size, i.e. to constrain WIP.

By limiting our WIP or batch size, we achieve the following beneficial properties:

- **Minimum queue size**
- **Minimum iteration length and cycle time**
- **Maximum speed to completion (Minimum time to market)**
- **Minimum technical risk**
- **Minimum personnel risk**
- **Minimum documentation**
- **Fast hypothesis testing, rapid learning**
- **Minimum cost of ownership per increment of completed work**
- **Reduce the number of blind assumptions that fuel our decisions**

Eliminating Unnecessary Work In Process (WIP)

Work in process goes far beyond the core development cycle. Here are some easy questions to identify wasteful work in process:

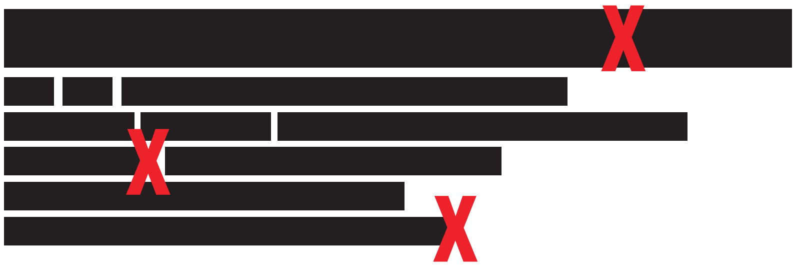
- **Are people creating documentation for products months in advance of any ability to implement?**
- **Do our designers over-design every interface and add features that we clearly do not have the capacity to complete?**
- **Do we have ‘brainstorming’ sessions in the middle of a pre-defined cycle?**

- **Do engineers over-engineer for their emotional gain? Does oversight exist to prevent this?**
- **Are all of our teams driven by the same organization priorities? Will they be able to cross the finish line together?**

Synchronicity Is Essential

With the absence of a framework for implementing constraints on a development process, things naturally get out of sync. This is very dangerous to an organization because upstream production will not respect downstream capacity. This relationship becomes very inefficient because different teams cannot unite around a common business goal. Furthermore, the collective parts of the organization become unable to share the responsibility of overall business success. Unsynchronized teams are prone to scary risk profiles, extreme waste, and hard failures.

An organization will end up with team-specific velocities, work in process, batch sizes and cycle times. If any other teams or the entire organization are dependent on a main large branch (or batch of team work), as seen below, the failure or misdirection of that main branch could be catastrophic.



A common example of synchronicity risk is the ‘swat team.’ Somehow the rest of a development team is either too busy or perceived as lacking enough talent to complete an initiative. A specialty team will be brought in to work in a vacuum, apart from the standard process. The failure points for this become misdirection, incorrect assumption and difficult project integration with the greater system.

Synchronize Small Batches Around Common Goals

The ideal scenario is for all teams in an organization to develop a predictable development cycle, with proper cadence and inter-team delivery. When upstream units respect downstream capacity, much of the unnecessary work in process is naturally eliminated, therefore creating more resources for revenue generating initiatives or optimizations.

CHAPTER 6: TOOLS & METHODOLOGIES

Any viable process methodology directly attempts to reduce batch sizes, reduce risk, improve transparency and predictability, and set a sustainable cadence to our development cycles.

Tools are just artifacts of our foundation

The main reason I love the philosophy behind lean development is that it is deconstructive; it's about the idea, not the tools. Teams and leaders often treat tool discovery as an epiphanic moment. However, because lean philosophy represents a framework for thinking and analysis, it is the deconstructive pursuit, not phenomenal luck or happenstance, that naturally leads us to tool discovery.

Agile, scrum, kanban, fifo, tdd, mvp, continuous integration, automation scripts. These are just a few words we hear every week in a modern software shop. They are all artifacts of trying to improve efficiency, reduce waste (time & labor), improve predictability, and reduce organizational (economic) risk.

Sometimes it's easy to get distracted by the details of a particular development methodology like Scrum or Kanban. How we use any really doesn't matter, it's all about generating economic results for the company and building a sustainable, ethical and respectful environments for employees.

One issue that I consistently see with development methodologies is that people have a hard time agreeing on exactly how they should be implemented. For example, I've never seen a Scrum process implemented the same way across two teams. That's because things like people, culture, technology and org structure dictate the extent to which things are possible.

Overview of some popular development methodologies for startups

Our chosen process methodology will be dictated by experience, org structure and desired outcome. Any viable process methodology directly attempts to reduce batch sizes, reduce risk, improve transparency and predictability, and set a sustainable cadence to our development cycles.

The same attributes that make software engineers so brilliant and focused are the same attributes that can cause efficiency disruptions in product development. Engineers live to solve the next problem, to construct the next beautiful virtual world, to use the next enticing framework or tool. This can be a major distraction and create problems for our business. Development teams need structure, priority and transparency.

Agile

Agile development is a general framework for thinking about process. It is best described by the Agile Manifesto, which made it's way online in 2001 and was drafted by Kent Beck, Ken Schwaber and 15 others. Generally, Agile describes a process for creating the shortest feedback loop through the following cycle:

- **Hypothesis (or Test for test-driven software development)**
- **Build**
- **Deploy & Learn**
- **Iterate**

Scrum

I have found Scrum to be the most effective process for teams larger than three. As teams grow, we need a process that can be decentralized and applied agnostically across groups with different disciplines. Teams naturally become co-dependent as they splinter. Scrum cycles (also known as sprints or timeboxes) allow separate teams to maintain the same delivery cycle and cadence, thereby minimizing synchronicity issues between teams, and associated organizational risk (as mentioned in the end of Chapter 5). Scrum also mandates that impediments are immediately removed or improved for team members, creating a more ethical and compassionate work environment. This point should not be overlooked.

Scrum has the following components:

- **Planning meeting**
- **Cycle or work iteration, bound by time (usually in weeks)**
- **Point system for work product based on developer effort (can be a proxy for estimated time)**
- **Daily standup**
- **Retrospective**

Let us use a one week cycle/release length as example. It is important to release the software from any Scrum cycle in its respective cycle. A team of QA and release engineers should not be responsible for finishing a release. The definition of done for developer tasks should be functional, deployed production code. If they cannot do this in a release, assign them less work. If there are organizational factors that prevent them from achieving this level of quality, those factors should be promptly removed or improved. For example, lack of automated configuration management, deployment systems or build pipelines.

** For anyone without an introduction to more sophisticated Scrum implementations, I recommend reading *Scrum & XP From The Trenches* by Henrik Kniberg.**

Step 1: Planning Meeting

Team leaders agree on product priority. Team members distill product components and volunteer for pieces of work by estimating level of effort. Because a team will learn its velocity, or production capacity of a cycle, they can provide a reasonably close estimate on each task. Individuals should prepare for this meeting in advance so no time is wasted. One of the most important parts of Scrum is that no member has the right to waste a group's time.

Index Cards

The planning meeting should result in task level work. The old-school way is to use index cards, with the name of a feature, or user story on the front, and all acceptance criteria on the back. The index card is the ultimate work constraint. If your scope of work for a given increment of work cannot fit on an index card, it's simply too big. This is particularly so with acceptance criteria. If acceptance criteria cannot fit on the back of an index card, distill the scope of work to end up with the minimum requirements for a demonstrable increment of customer valued work.

An excellent book about user stories and acceptance criteria was written by Mike Cohn, called *User Stories Applied*.

Step 2: The Sprint, also called the Cycle or Work Iteration

After the planning meeting, the team is ready for its sprint. The length of the sprint = one cycle. As discussed earlier, the total work underway at any time during a cycle is called WIP. One thing to note about batch size compression as it relates to Scrum, is that we should strive for a reduction in batch size at any level of magnification. For example, each task that a developer agrees to build in a cycle should represent the smallest demonstrable unit of customer value, ideally no more than 4-8 hours of uninterrupted work. Moving one level higher, we should strive to keep the team's overall sprint or cycle length as short as possible (ideally one week).

As a team, because we were prepared with disciplined priority and distilled ideas, once a cycle is started, no single member at any level of the organization may interrupt it. If the CEO wants something, he can't get it. If the product manager changes her mind, she's out of luck. It is the important job of the team leader or scrum leader to block all external interruptions or distractions for the team. It is also the team or scrum leader's responsibility to remove any and all impediments to the team's work that arise during a cycle. If a team is working in one week cycles, the maximum amount of risk we can generate is four days of labor. That is a very small amount of risk distributed across given our disciplined preparation.

Just as important, interrupted cycles cause three problems: engineers lose their momentum and never regain it, managers lose their authority, engineers feel slighted. It is disrespectful to provide engineers with improperly prioritized business and feature goals. They are not pawns to be moved about without proper planning.

Step 3: The Daily Standup

The daily standup is one of the most important components of Scrum. In fact, it closely represents the name of the methodology. Each morning team members gather together for a quick status update, or scrum, before continuing on. Just as in rugby, the team has the overall goal of winning. However, synchronicity is key. The match moves in small increments of intense activity. If half of the team members are oblivious to the movements of their teammates, insufficient work product will be produced to achieve victory. It's akin to a rugby flanker trying to score a tri without the ball.

In each daily standup, we report the following, with absolutely nothing extra:

- **Am I blocked on anything?**
- **Do I need help from a team member on something?**
- **What am I doing today?**
- **If absolutely essential to tell the team (which is rare): What did I do yesterday?**

This method of daily standup should take no more than three minutes for a team of six people. No one should be allowed to talk about personal matters, strategize, or get off course. All further discussion should be taken offline. People should always be discouraged from wasting time in a group setting, it is extremely costly. The daily standup is not a place to practice public speaking. Future lobbyists, senators and thespians should practice in front of the mirror at home, not in front of the team.

***The standup should always be done standing. No sitting. No leaning.**

Step 4: Retrospective

This should be treated like a standup. Address any issues or interruptions of the past cycle, and any improvements the team can make in the upcoming cycle. Take all other conversations offline.

That's it. Simple. Scrum is also convenient because employees work Monday-Friday, and that length of time can provide a nice timebox.

Also, timeboxes provide a visual and literal construction for an incremental release of finished product.

Kanban

Kanban is a simple system of modular process constraints. There are three steps associated with Kanban:

- **Queued or Ready**
- **In Process**
- **Completed**

The goal of Kanban is to limit the amount of work in process, also known as WIP. For example, each team or developer may only have two in-progress tasks. This method of Kanban is the most simple process constraint you will find.

I don't care for Kanban because it is not accountable to a timebox and loses the cadence and rhythmic cross-team achievement found with Scrum. Furthermore, caps on the number of ongoing items become quite nebulous, and team members are less incentivized to properly estimate work without a structure for consistent weekly milestones and deadlines.

However, Kanban can create a beautifully simple FIFO (first in, first out) system. I suggest using something like the Simple Team Principle (discussed in the next chapter), combined with point or time based constraints on all work in process, as opposed to feature or project-based constraints on in-process items.

CHAPTER 7: THE SIMPLE TEAM PRINCIPLE

Regardless of team structure, if we use each other to enforce simple thinking, we get to the core of lean without all the fuss.

“Keep It Simple, Stupid.” These words have been offered by uncles and grandfathers the world throughout. They are certainly keying on something very powerful.

By keeping things simple, we achieve the following beneficial results (as seen in Chapter 4):

- **Minimum work in process**
- **Minimum queue and batch size**
- **Minimum iteration length**
- **Maximum speed to completion (Minimum time to market)**
- **Minimum technical risk**
- **Minimum documentation**
- **Fast hypothesis testing, minimum cycle time, rapid learning**
- **Minimum cost of ownership**
- **Reduce the number of blind assumptions that fuel our decisions**

The Simple Team Principle

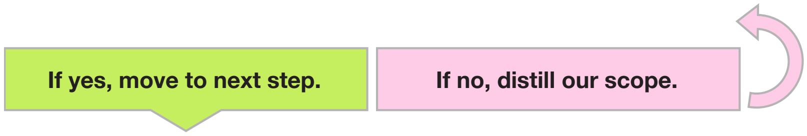
I would like to propose the Simple Team Principle as something every team can use as the foundation for lean decision making. Most of the teams we work with in fast-moving software shops are between 2-8 people. Some teams are cross-functional (dev, UI, product, exec). Some teams are problem-specific (API team, data warehousing team, frontend team). Regardless of team structure, if we use each other to enforce simple thinking, we get to the core of lean without all the fuss.

To begin, your team starts with a high-value product idea that leaders in your organization agree to test. First, that idea should be deconstructed and distilled to its essence. What is the part of the idea that will generate the highest value? Agree to test that first, learn from your data, then repeat.

1. Simple Scope Agreement

What is the most simple scope that will lead to the successful validation of this idea? Generally, this scope proposal should be non-annotated wireframes or diagrams. I prefer drawing on a scrap sheet of paper or a whiteboard first, then bouncing the idea off others for a specified amount of time, like 15 minutes max. Then move to more formal wires without annotation. That should be sufficient for agreement. Distilling the scope on paper first is very important. Always ask yourself, do I really need x,y and z to create a releasable product?

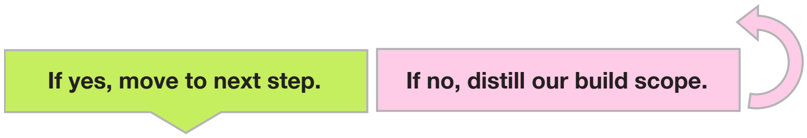
Does everyone agree that we have chosen the most simple scope that can lead to the validation of our idea?



2. Simple Build Agreement

How can we technically build or validate our idea in the most simple way? We want to minimize cost of ownership, promote easy maintenance, reusable components, etc. If we are wrong, the cost of starting from scratch should also be low. Can we use an external service to test our idea? Has someone already built the tool to help us build and test our idea? The list of possible service providers is becoming limitless.

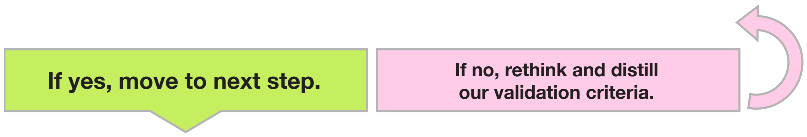
Does everyone agree that we have chosen the most simple build path that can lead to the validation of our idea?



3. Simple Validation Agreement

What is the minimum amount of data we can gather to validate our idea, and how will we get that data? How many installs, visitors, purchases, leads, sales, contracts, referrals? All measures of data should back out to a raw dollar figure. For example, we agree to spend $1,000 for in-app purchases. If we make back $500 in 30 days, we think we can optimize further and that our customer LTV will naturally increase. As another example, sometimes the validation is time: delivering a working piece of software using the tools decided on above, in the minimum (and promised) amount of time. Time can always be valued in man-hours, man-days, date of release and revenue per day, etc.

Does everyone agree that we have chosen the most simple validation requirement for our idea?


If yes, move to next step.
If no, rethink and distill our validation criteria.

We Have Completed One Iteration

Assuming we have proven that our idea is valid, every next step should be an iteration of demonstrable progress. Each next step should be accountable to the requirements above. Forward progress and experimentation should always be bound by simplicity. It should not matter if you're building out a massive sales funnel, a sophisticated deployment process or buying installs for an app.

Furthermore, any process tools that you decide to incorporate can be treated like an idea or product and bound by the Simple Team Principle. For example, you want to use an agile tracking software. Ask yourself all of the same questions above:

1. **Simple Scope Agreement. First, why do we want to do this? To improve transparency and accountability. What should it be able to do? Show and track tasks, email us, integrate with git and our build server. Must support inheritance and bundled 'cards.'**
2. **Simple Build Agreement. Is building this ourselves worth the investment? No. What is on the market that serves our needs? Will this product be easy to maintain over time? Can this product grow with our team?**
3. **Simple Validation Agreement. We agree that we cannot risk disrupting the entire team with a hard change. Using our platform of choice, two people will model six example workflows for two weeks alongside our current process. If we are satisfied with the results, we will iterate to implement our chosen platform with the entire team.**

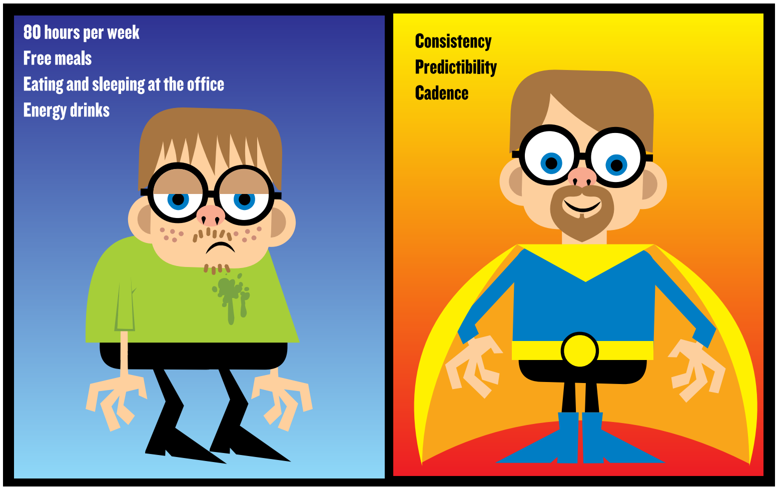
CHAPTER 8: STRUCTURING TEAMS FOR SUCCESS

Consistency, predictability, cadence. These things carry our team over time.

Building Ethical Systems

Talk to most investors and founders, and they will advocate for a relentless employee culture of working 80 hours per week, free meals, eating and sleeping at the office. Where did we acquire this mentality? It is completely unethical. If our engineers or product developers need to work 80 hours per week to achieve success, maybe we should rethink what we are building and re-prioritize their workflow.

Consistency, predictability, cadence. These things carry our team over time. Teams that frequently work through the night fueled by the latest energy drink are prone to erratic and emotional behavior, unreliability, and health crashes. Furthermore, if we encourage developers to keep a healthy work-life balance, they will surely use some of their free energy to solve problems and create efficiencies. Their brains will also solve our problems with less fatigue. They will operate within any constraints our culture mandates.



Confidence in a well-structured process is essential for long-term success.

Team Structure

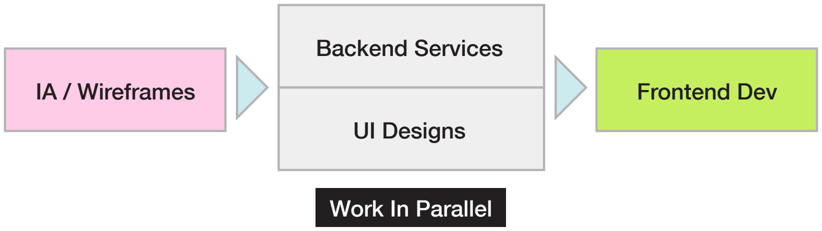
Most holistic software development teams are comprised of the following functional components:

- **Information Architecture & User Experience/Workflow Design**
- **Backend Development**
- **User Interface Design (graphic design)**
- **Frontend Development**

Most importantly, all essential functional design questions should be answered on paper, at the diagram or wireframe level. This is the least expensive way to develop and distill ideas.

Second, it is always more efficient to separate ‘function’ from ‘fashion.’ When we combine UI design with functional development, the cost of change becomes much higher.

With our ideal structure we end up with a pipeline of planning, parallel backend development and visual design preparation, and finally the ‘skin’ or frontend development that can include complex javascript and interface components.



This structure may seem like it sets up gates and blocks. However, because our cycle time or iteration length is as short as possible, ideally one week, there is a maximum of one cycle of waste for Backend Services / UI Designs. By the time a feature gets to the frontend team, it should change very little.

CHAPTER 9: SKETCHES, WHITEBOARDS & WIREFRAMES

It is our essence, or the big idea,

that is both extremely valuable, but

can also be developed and shared

at an extremely low cost.

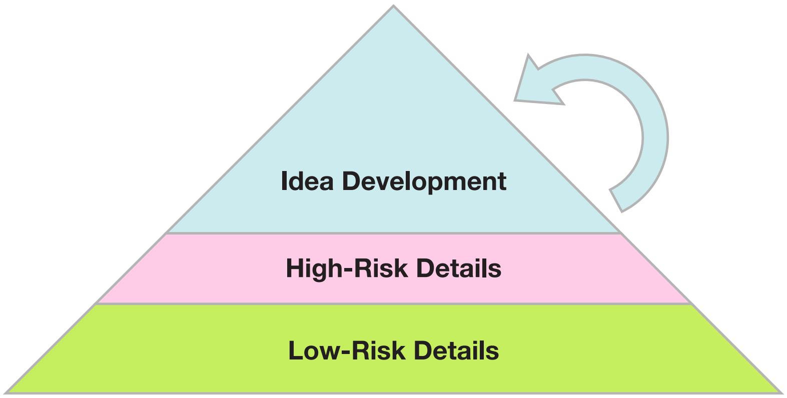
Much like mining for gold, so should our processes sift out large chunks, top to bottom (in our case, large chunks of risk). As discussed previously, if we can eliminate maximum waste early on in a process, our overall cost savings can be phenomenal.

Iterating through ideas in functional code is very expensive. It lends itself to high man-hour costs, high cost of correction and increased technical debt. As such, any complexity evaluation we can do on paper (or a whiteboard) is essential. When we say paper and whiteboards, we want visuals, not novels. Words and paragraphs are a very poor representation of software. Workflow diagrams and wireframes are much more efficient. Our idea development process should not require traditional written documentation.

Afraid to Show Our Progress

Throughout my career, I have noticed that people rarely like to show people their ideas until they are 'ready.' Unfortunately 'ready' is a nebulous and subjective state of being, often representing a large batch of work that others review and say, "Wow, this probably took you a lot of time…" Who cares how much time was invested if the underlying ideas are misdirected?

With lean, we want early information to influence our direction. Sketch it on a piece of cardboard, or a napkin, or a whiteboard and share it with your teammates. Most of the value we create is in a larger idea, then we fine-tune with details as needed. If we cannot 'sell' the big idea to people immediately, there is a problem.



It's like building the most beautiful boat in the world, except it sinks in the marina.

- **What kind of beautiful wood shall we use?**
- **We need bright sails with awesome logos "Dr. Adventure, M.D. Ph.D. MBA"**
- **Oh, and we need a great radio system**
- **Top of the line GPS and nav, check**
- **We're going to want to fish off this boat too, definitely**
- **Ok, does it have a large beer cooler? Perfect.**

Oh, wait, we forgot to make sure this beautiful vessel can float. Just like with boat building, misdirected investments in complex software are rarely recoverable.

Do It Like Picasso

There is a famous story where Pablo Picasso pays his bar tab with a napkin sketch. Was it what we know of as a complete work of art?

Probably not. However, given the circumstances it was the minimum investment for the maximum return. Just like the tenured product developer, Picasso was an expert.

If we are confident in our skills, we should approach things the same way. It is our essence, or the big idea, that is both extremely valuable, but can also be developed and shared at an extremely low cost. For the bartender, it was the right product. In our example of idea generation and distillation, our teammates are the bartender.

CHAPTER 10: FINANCIAL MODELING

When asked how much the young entrepreneur needed to raise for one year of operational capital, his answer was a confident $1.5M.

Simple financial modeling is often overlooked as one of the most helpful instruments in understanding the potential value created by a proposed new product. To use a real-life example, I recently had coffee with a young Internet entrepreneur. He was working on a startup targeting the crowd-funding space that would provide a ‘freemium’ value-added service to existing crowd-funding projects. He estimated the total market of crowd funding projects to be around 100,000 projects per year. Ok, great.

When asked how much the young entrepreneur needed to raise for one year of operational capital, his answer was a confident $1.5M.

So we target that 100,000 potential customers and let’s say we convert 5% into paying customers. We have 5,000 paying customers. Because crowd funding projects are usually singular events, he suggested that each paying customer will net us $50 per year. We will make $250,000 in top-line revenue per year. After we reviewed the numbers in our head for a few moments, he agreed that a net $1.25M operating loss each year might not please his investors.

The example above is incredibly common. As product developers and entrepreneurial types, the lure of the pursuit, of building something exciting, often causes us to lose track of important details. There have been many times in my career where I’ve had an idea that was romantic and alluring, however after modeling out the entire picture, the idea no longer seemed viable at all. Small details can change one’s entire perspective.

As we move forward with any idea, it's easy to create a simple financial model to analyze how the actors in a system affect our bottom line. Any thorough financial model should address the following (at a minimum):

- **Time Increment (monthly or quarterly)**
- **Gross Revenue**
- **Cost of Goods Sold (COGS)**
- **Cost of Selling (also known as SG&A)**
- **Addressable Market**
- **Market Penetration**
- **Conversion Rate**
- **Cost of Customer Acquisition**
- **Employee Expense**
- **Available Cash Balance per Time Increment**

Below is a generic table that illustrates the above information.

	Jan	**Feb**	**Mar**	Total
Summary				
Gross Revenue	**$700**	**$2,800**	**$6,300**	**$9,800**
COGS	**$(140)**	**$(480)**	**$(900)**	**$(1,520)**
Cost of Selling	**$(24,700)**	**$(27,600)**	**$(30,900)**	**$(83,200)**
EBITDA	**$(24,140)**	**$(25,280)**	**$(25,500)**	**$(74,920)**
Revenue				
Addressable Market	30000	30000	30000	
Estimated Tangible Customers	2000	4000	6000	
Conversion Rate	1%	2%	3%	
Total Customers	20	80	180	
Average Order Value	$35	$35	$35	
Total Revenue	**$700**	**$2,800**	**$6,300**	**$9,800**
Cost of Goods Sold				
Cost of Computing Per Customer	$(7)	$(6)	$(5)	
Total COGS	**$(140)**	**$(480)**	**$(900)**	**$(1,520)**
Cost of Selling				
Cost Per Customer Acquired	$(60)	$(50)	$(40)	
Event Sponsorship	$(300)	$(400)	$(500)	$(1,200)
Total Marketing Cost	**$(1,500)**	**$(4,400)**	**$(7,700)**	**$(13,600)**
Employee Expenses - Tech	$(20,000)	$(20,000)	$(20,000)	$(60,000)
Employee Expenses - Customer Service	$(3,000)	$(3,000)	$(3,000)	$(9,000)
Web Hosting & Related Services	$(200)	$(200)	$(200)	$(600)
Total SG&A	**$(24,700)**	**$(27,600)**	**$(30,900)**	**$(83,200)**
Beginning Cash Balance	$300,000	$275,860	$250,580	

CHAPTER 11: KEY METRICS & SUCCESS MEASUREMENT

Team measurement should

always be closely aligned with

organizational business priorities.

Metrics and Validation Analysis

As learned product developers we must operate under the following assumptions:

- **Everything has a value**
- **Everything can be measured**
- **Time costs money**

An important place to start is with the declaration that we must be able to answer any question that might arise. But what to know and why, that is the most important thing for a startup. As any data-driven startup will tell you, data can quickly overwhelm us until we have more data than we can reasonably act on. How do we distill all of this into actionable data?

Perhaps a nice place to start is to have everyone on our team identify their most important economic performance and waste indicators. Those indicators might be aggregated into a dashboard. For example:

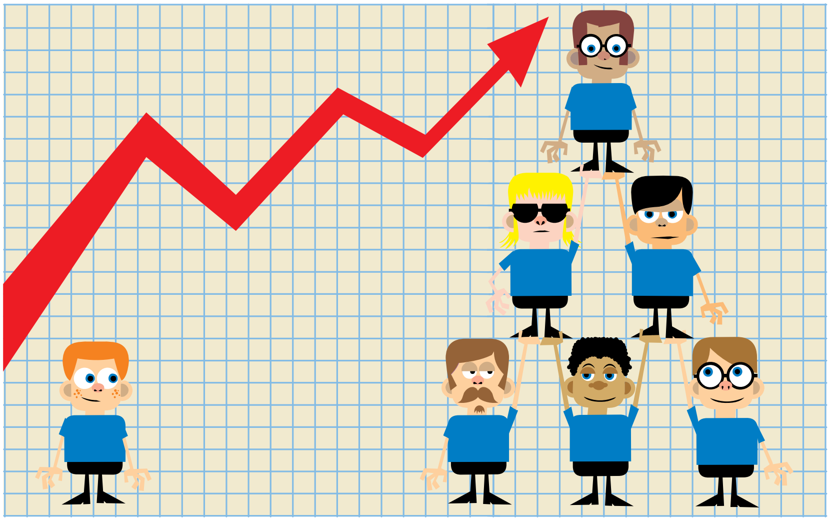
- **CTR on emails**
- **Purchase rate from emails**
- **Unsubscribes on emails**
- **Conversion rate through each step of a funnel**
- **Code Rollbacks**
- **Cost per install per channel**
- **Aggregate cost per install**
- **Shares / Uniques**
- **Pageviews / Uniques**
- **Return Rate**
- **Chargeback %**

- **Points / Player**

The above dashboard example is yet another candidate for the Simple Team Principle.

Team Measurement

Team measurement should always be closely aligned with organizational business priorities. Did product dev or tech or customer acquisition provide for the rest of the organization when required? For example, was marketing empowered by tech, was tech empowered by content, was physical product empowered by operations?



Some technology teams focus on agile tracking velocities and burndowns. At the end of the day nothing matters except for delivering on time for the business in micro cycles. Again, a reason I favor the weekly iteration length or sprint cycle. Long burndown charts and aggregate point tracking over time imply long release cycles -- as fast moving product developers, long release cycles come to represent relics of everything we are fundamentally against.

Supporting deadlines and customer-facing deliverables is also very important to retain team urgency. Because we respect our employees and have encouraged them to enforce structured development processes, it will be very difficult to over-stress the team in the presence of important business deadlines.

Micro Experiment Testing & Optimization

Optimization is very important over the life of any product. Companies like Amazon, Google, Facebook and Etsy have proven this. When we launch a product, we must always be prepared to invest in incremental improvements to the product. There are a host of services that can help with this. Again we focus on short iteration length and small batch sizes. In essence, each release is an experiment. Perhaps next week's priority is optimizing a previous product version. Perhaps we find ourselves in dire straights, and must experiment in a more drastic fashion. Whichever scenario we encounter, so long as we know that our risk profile is low, we set the table for success and rapid learning.

CHAPTER 12: WHAT MAKES A GREAT DEV TEAM

Often we hear of teams looking to hire 'hackers' and 'ninjas.' Experienced, prideful, professional engineers are not hackers.

Our team is everything. As such, lean practices should be mandatory. Lean allows teams to consistently feel the satisfaction of shipping products to customers with associated learnings. Lean allows teams to stay on point with the most relevant data.

As we covered earlier, people can greatly influence the total cost of ownership of a system. As we create our blueprint for personnel and hiring, it is essential to keep this in mind. Hiring managers often feel underwhelmed with the level of available talent and end up compromising in order to move forward. While this is quite a reality of our industry, it is very important to remember what our ideal profile is. Even if we have to compromise over the short term, we should always try to 'hire up.' For example, we hire an engineer who we do not feel comfortable promoting into a management position, that's ok sometimes. Unfortunately that person might feel compelled to look for a new job, but we cannot compromise the integrity of our long-term vision.

As you look to hire the following positions, I offer some points to keep in mind. After years of recruiting and hiring, I don't like to compromise. I don't feel it's necessary. If we build the right kind of systems, we will naturally attract top talent. If we follow that same spirit for many years, then we end up with a strong network of like-minded individuals. Hiring becomes much easier.

Product Managers

The product manager title is one of the most nebulous and hijacked positions in all of software. Software development is one of the few industries where something like a person who has never written a line of code can lead a team of code writers. For example, I cannot show up on Wall Street with an MBA and convince everyone that I am fit to be managing all trading for a hedge fund. I cannot show up at a Local and decide that I'm fit to lead the next round of industrial welders working on a suspension bridge. Somewhere along the way, our industry has forgotten about craft, mastery, apprenticeship.

It is impossible to understand the nuanced economic cost and risk associated with proposed new products without having built software in the past. People who have built real software create organizational efficiencies in the way they communicate and remove excess waste early on in a development process.

Talk to any engineering team and they will tell you that a consistent cause of headaches are product proposals that lack the specialized understanding needed to operate within a defined time and labor scope.

Analytical thinking and an MBA are just not sufficient, I want people who have PROVEN with real working product that they understand software. Just because someone refreshes TechCrunch five times a day and can talk the talk about Agile processes does not make them fit to lead and/or be the upstream resource for a team of engineers.

If innovative or high-level thinking is what you're after, it might make sense to frequently pair a highly operational and experienced software builder with consultants who can provide more innovative strategy at important times in a product's lifecycle.

Experienced Engineers

Senior level engineers are said to be up to 10x as productive as junior level engineers. If we agree that time to market, minimum technical debt, and the smallest risk profile are of paramount importance to the business, junior developers do not make economic sense.

Often people reference Google or Facebook as hiring any undergrad engineer and paying them $100k+ just to show up. What those same people will never mention is that Google and Facebook are 1000+ person organizations with the capacity to properly train engineers. Training young developers takes years, not months. Small teams will have little capacity for training people and enforcing the rules while moving at light-speed.

In senior engineers, we want prideful individuals who feel it is their responsibility to write production quality code. As members of small teams they do not need QA engineers because they surround themselves with the tools to automate the testing and deployment of their code into production.

Ultimately, quality engineers and personnel are attracted to a well structured, professional, efficient and ethical environment. Often we hear of teams looking to hire ‘hackers’ and ‘ninjas.’ Experienced, prideful, professional engineers are not hackers; they are architects, masters of mission critical business systems, leaders in application availability, throughput and performance. They have served people over a billion ad impressions, they have built real-time gaming systems, they have powered billion object search systems.

These quality people will not work for amateur dev shops, that's why they are so hard to hire. Build a great process and you will naturally attract high quality talent. It's a spiritual trend, like mountaineers drawn to the range.

Systems Engineers

Automation, automation, automation. If your systems people do not employ tools that provide 100% automation, you're in for a long and painful ride to the top. In the era of smaller, virtualized server environments, your team will have to manage a lot of instances at any measure of reasonable scale. Sure, your systems person with experience racking machines might be amazingly efficient running three super-powered web servers. However, always consider how well they will handle 120 machines across four tiers of development environments, often with each tier containing more than five different configurations for things like web server, cache server, database, search system, queue server, etc. This kind of setup is grossly inefficient to maintain without proper automation capabilities.

Hiring Personalities

Never compromise and hire people because they seem smart, while at the same time you recognize deficient social or interpersonal behavior. Your team members will spend some of the most intense and focused time of their life together. No matter how smart someone looks on paper, people with personality issues who seem confrontational, stubborn, narrow minded—coincidently, they build similar types of systems—inflexible and fragile, incapable of adjusting to an environment that is dynamic and unpredictable. In an environment where tool use and flexibility are paramount, there is rarely a need for the hero or ‘artist as genius’ in an engineering organization.

CHAPTER 13: GO MAKE A CUSTOMER HAPPY, TOMORROW

As we make our systems more efficient, we are able to 'listen' to our customers more often. They prove to us what they like.

In closing, what lean really teaches us, as illustrated by the wholesale failures of young startups, is that our software startup industry is rooted in risk and assumptions. We need frameworks for thinking that systematically reduce the assumptions that drive our decisions and growth, thereby reducing the risk we encounter while trying to innovate and incubate new products. The tools we find along our path are just artifacts born from continually trying to improve the way we build and deliver working software.

Never has there been a more exciting time for a product developer or entrepreneur. They call our funding “Adventure Capital” for a reason. As our suite of tools becomes ever more commoditized, compressing our time to market becomes easier and cheaper. Never before has it been so easy to experiment with business ideas.

We are on a learning adventure with our coworkers, but most importantly, with our customers. As we make our systems more efficient, we are able to ‘listen’ to our customers more often. They prove to us what they like with their attention or their hard-earned money.

CHAPTER 14: RANDOM MUSINGS

If you think you're moving quickly, try 10x harder. Strive to find the limits of every part of your organization and refuse to be afraid of failure.

Good companies are always in Beta

Many companies consider their Beta period to be a time of uncertainty and idea validation. Strong companies are always in Beta. They are always running experiments at every level of magnification to learn more from their customers.

Advice for Small Startup Teams

Ok, so you have your ‘big’ idea, we get it. And congratulations, you somehow tricked your investors into believing you can build this amazing construct in four months.

Now it’s time to start thinking small. Focus on what you can complete today. While you’re waiting around for business deals, experiment with more of your hypotheses. The cost of experimentation goes down every day. Ship something to the customer tomorrow if you can.

If you think you’re moving quickly, try 10x harder. Strive to find the limits of every part of your organization and refuse to be afraid of failure. The speed you’re afraid of is what separates real growth from just another competitor. Also, assume there is a tool on the market for automating almost every business process. Look for places to implement automation whenever possible.

Advice for Small Teams in Large Organizations

Set an example for the rest of the company. Compassionately educate others upstream from you. Inspire other teams to eliminate waste and improve efficiency.

Advice for Product Managers

Think like an agile software developer. Always match the cadence and throughput of your engineers. Don't plan and design excess product inventory that no one can build for months. If your business operates at any respectable pace, something will happen next week that changes the way you think about the world, rendering your 'plans' useless. Always thoroughly research the products offered by service providers as they have likely solved a problem that takes years to master. Through them you may be able to gain a huge competitive advantage via rapid time to market.

Advice for Engineers

Support the business. Stay away from shiny gold ideas as they are almost always a quagmire. Your distraction can turn into serious failure in a heartbeat. Remember that someone out there solved a problem 100x the size of yours with what many consider to be unsexy, commodity technologies and frameworks. Use their learnings to your benefit. Stay away from unproven technologies; when your business grows rapidly, it will be infinitely harder to hire. Furthermore, it's difficult to predict the failover qualities of new system components that have not yet been stressed in your type of application environment. Ok, so you became Facebook or Instagram? Congrats, now you have real (unique) problems and beers are on me.

About the Author

William Belk's core focus centers around operational incubation, Lean process improvement, product design/strategy, user experience and complex data problems. With over a decade of early stage startup experience, William has served in roles ranging from founder, investor, first-employee, mentor, community organizer, strategist and advisor.