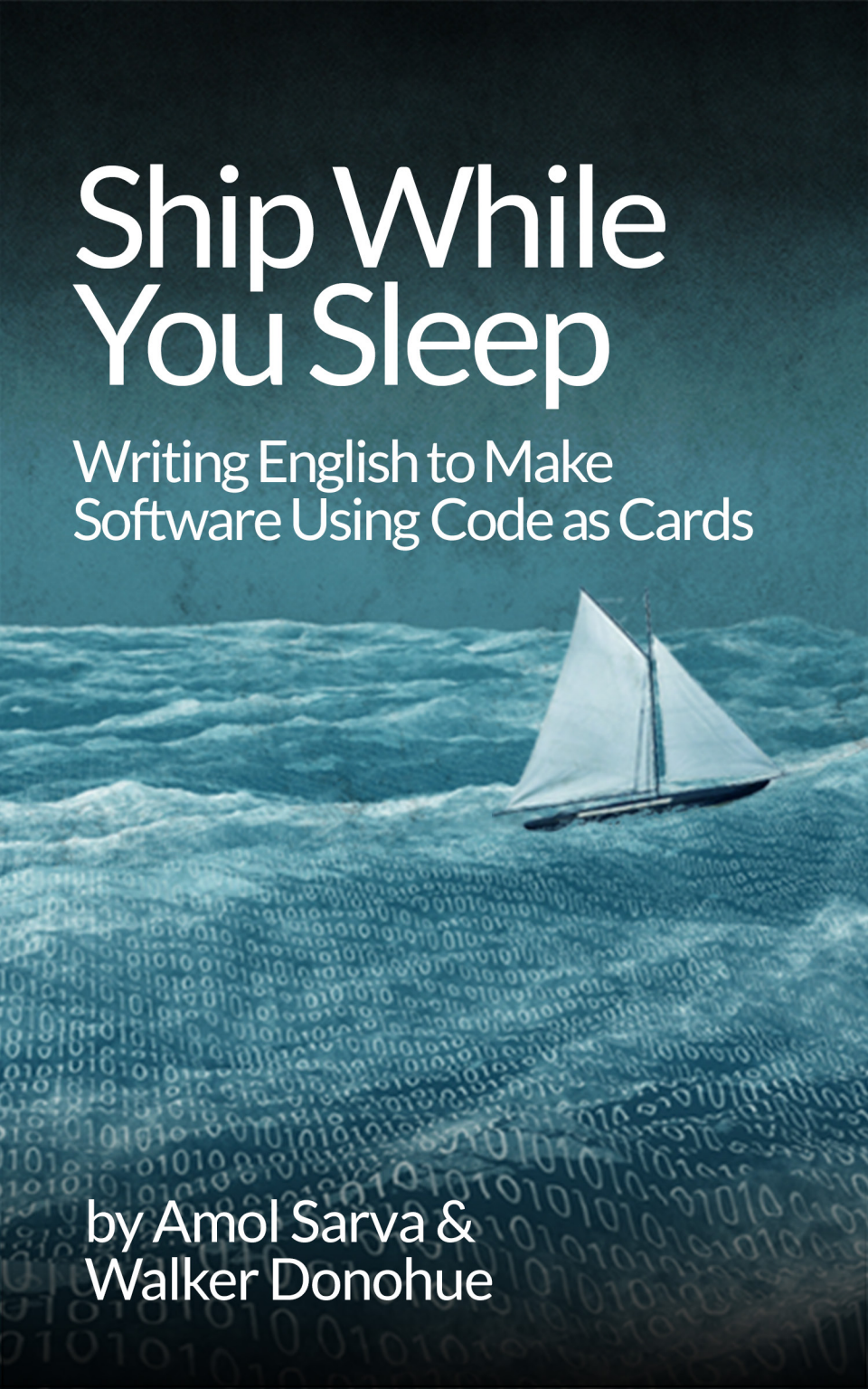
Ship While You Sleep

Writing English to Make Software Using Code as Cards



by Amol Sarva &
Walker Donohue

Ship While You Sleep

Writing English to Make Software Using Code as Cards

By Amol Sarva &
Walker Donohue

About Amol

This book on hiring and building is based on some experience. In 1995, I worked as an intern learning Java at one of Fred Wilson's early investments (I was a hobbyist hacker and philosophy major at Columbia in New York). I started my own little startup putting NYC menus online in 1996. In 2000, while I was finishing a PhD at Stanford, I helped start Virgin Mobile USA - a huge business that raised $500 million in venture, hired a team of 100s of engineers and went public.

Since then I've started 10 more companies and invested in 25. Some of the better efforts included Blue Mobile, where we raised $20 million in backing to make another mobile phone service in the US with Wal-Mart and Verizon as partners (less cool than Virgin) with nearly 100 engineers at work; and Peek, where we made a Blackberry-type smartphone that cost 95% less to make. That was a real feat that won us some awards like Wired's #1 Gadget and Business Week's Gadget of the Year

and became a software business whose US unit was acquired by Twitter and the global platform acquired by Bharti SoftBank.

Since 2012 I've been part of building a litany of products with large startups and new startups -- a streaming music player called Gramofon, a wearable neurostimulation startup called Halo Neuroscience, a new kind of team workspace called Knotable. Plus 1-2 person teams building mobile apps and public companies deploying our products into $100 million revenue customers.

Then there are the dozens of companies where I'm an investor or advisor -- Klink (smart caller ID for sales teams), Kollabora (a crafter and maker community site), Ouya (the open source hackable Android game micro-console), Social Bicycles (the self-organizing bikeshare network for cities), KeyMe (the key-copying robot and smartphone app), Plethora (the manufacturing robot), FYT (the personal trainer on demand marketplace), Caliber (the app where professionals can meet new people), WorkMarket (the platform for managing freelance workers), MarleySpoon (the dinner recipe in a box delivery service), and more.

The way we build and ship software products is changing a lot -- the next step is already visible. As Alan Kay said "The future is already here. It's just unevenly distributed." In this book I lay out a way to work and ship that ties together a dozen trends we are seeing in how the cutting-edge organizations are making things. It's a recipe. But in the recipe you see what the future of work will look like.

Chapter One:
Beyond Code as Craft

The team of engineers at Etsy, on their blog Code as Craft, espouse a method of software development they compare to the way a cathedral might have been constructed in the Middle Ages:

> *"Each took thousands of person-years of effort, spread over many decades. Lessons learned were passed down to the next set of builders, who advanced the state of structural engineering with their accomplishments. But the carpenters, stonecutters, carvers, and glass workers were all craftspeople, interpreting the engineering requirements to produce a whole that transcended the purely mechanical side of the construction. It was their belief in their individual contributions that sustained the projects: We who cut mere stones must always be envisioning cathedrals."*

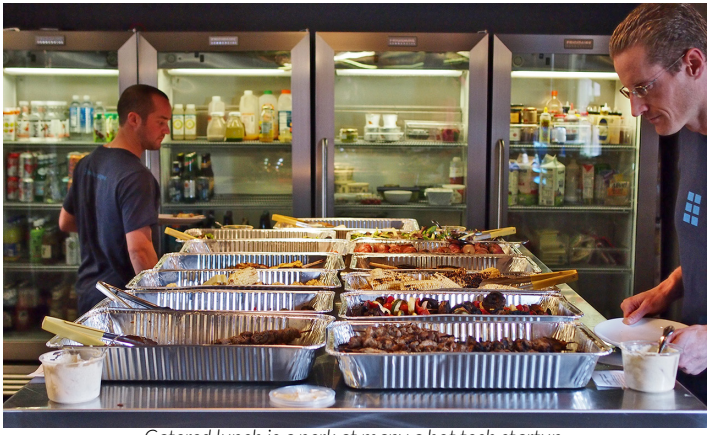
It is an amazing visual. In many ways the uber-hackers of today closely resemble the master craftsmen of the 1490s.

Long beards, skilled hands, makers, artisanal and rather than commercial. The cultural cousins of these people are making chocolate by hand in Brooklyn, or writing novels in longhand. They believe in the supreme, divine power of genius and wisdom. They make masterpieces of code.

There is a small irony in that passage about cathedral-craft, and it concerns a famous 1990s paper by Linux and open source theorist Eric Raymond. Back in that era, the enemy was Microsoft and their ponderous operating system Windows, and Linux was thought to be the wave of the future. The monopolists in Redmond made an operating system so good and so carefully designed to work on so many computer makers' machines, that many corporate and consumer technology buyers believed nobody could ever catch up.

It was at this time that the open source movement was gathering steam, around an open source operating system that nobody would own or direct: Linux. Led by a loose band of hackers all over the world, the Linux codebase was the Wikipedia-style global project of its day. Nobody in particular entirely in charge, no top-down architectural edicts, no dedicated full-time army of engineers laboring away.

Back in the 1990s, the idea that a Linux operating system would one day conquer the world was silly (it was also correct; more computers now run Linux-based OSes than Windows by a large margin). And the guy who most famously argued for why this might work out used a simple metaphor. In a paper called "The Cathedral and the Bazaar", Raymond more or less advocates a marketplace of ideas, the specialization of



Catered lunch is a perk at many a hot tech startup.

different tasks, and the continuous improvement of an iterative approach. Quite the opposite of the grand cathedrals.

So it's a clue: something in the current engineering culture has pulled people way off the path of a powerful, crowd-sourced, open-source, Bazaar ethos to a new form of guild-making craft.

That's all well and good. But say you have an idea for a web app: for example, a site that will let people essentially start their own businesses by uploading their wares and selling them to whoever is interested. And you need to develop this idea, and you hold the philosophy above dear to your heart. How do you motivate engineers to see themselves as master craftsmen? How do you hire these master craftsmen in the first place – away from their well-appointed guilds and long-term artistic crusades?

According to a thousand profiles of tech companies

(e.g. this gem from Bloomberg Businessweek), you give new employees stipends to decorate their offices, hire a chef to prepare meals with locally-sourced ingredients, and let dogs roam around (how cute!).

Or look at Rap Genius, mentioned in the same article: every new employee there gets $1,000 a day, can take unlimited vacations, has access to unlimited free Seamless and Fresh Direct, as well as a shower, a gym, and a laundry room. At bigger tech firms the luxury is famously lavish too: Michellin-star chefs prepare meals at Google's cafeterias around the world.

From the business side of things, very, very few people with ideas for apps have the kind of infrastructure and money required to support this kind of work: the coddling, luxury style that so many hip new start ups seem almost obliged to provide if they're going to attract talented work.

Same for established companies with a little less go-go money-to-burn. Maybe at your company you have a project you think should happen, but you don't have the budget to go headhunting for hotshot CTOs or perhaps even internal engineering staff. How do you get it going? Many of the problems a startup founder faces translate directly to those of a larger organization.

Starting at ground zero, what's the first step to turning your mock-up for an app into a real, shippable product? For many in the startup world, it's finding a co-founder, someone who understands the scope of your project and can do the work of coding it. Which raises the question: how do I find that

person?

Wolfgang Bremer, German designer, entrepreneur and co-founder of an online service for (the irony is impeccable) finding co-founders, says, "I'm constantly going to these startup events, meetups, and tech talks... however, I often notice that always the same people seem to go there. And they're going there to find a co-founder for their own idea. So very rarely there's actually a potential co-founder available. And that sucks."

It sure does. It's really easy to see this first-hand, especially if you live in New York City. You can even go to parts of Brooklyn that were basically war-zones in the 1970s, and today, the sidewalks outside bars are full of people on Friday and Saturday nights scoping out potential partners for their new idea. Why bars in particular? Three reasons. One, because that's where like-minded people congregate, and where you might be more likely to find your perfect fellow traveler. Two, because start ups need to be cool, right? And three, because people in bars are tipsy and susceptible.

Or even consider how teams you have already assembled manage to work together. Engineers start working, coding away, headphones on, eyes on screen. "Meetings suck," they say. They roll their eyes at management types. They argue with business leaders about what the product should do and how, or even who the customers are.

Partly, engineering culture has embraced a solo hacker culture where people collaborate at their own discretion. No hierarchy. Only grudgingly taking directions.

And partly, engineering culture today has gotten deep into the supply-demand imbalance of the global tech market. Engineers are scarce. They are prima donnas. They do what they want. CEOs fear them.

It's an expensive worldview to adopt. If you are starting a company you need to find this hacker co-founder - not simple if you are searching for a Code As Craft master. If you are launching an internal project, you need to plead with the CTO-organization for the precious time and interest of their All Star Team - a group of people who normally wouldn't bother to say hello to you in the cafeteria. And if you are running a large engineering organization, reasonable efforts to be flexible and productive leave you constantly skirmishing with an irritated Mob of Villagers who hate "suits".

Why is this happening? At the heart of the issue is simple supply and demand. In rich places like the US and Europe, there is huge appetite for software engineers and limited supply. Immigration laws are constraining, and so where venture capital, startups and technology projects are plentiful, business managers just cannot get enough people. So the competition for people takes all forms: high compensation, risk-averse engineers, bad attitudes among teams, high churn among staff. Like any system with a scarce few stars, you start seeing poor behavior by the overpaid, irreplaceable franchise players.

It's such an imbalance that many early-stage companies cannot raise money if they don't already have a compelling engineering organization. Nobody believes you will get one when you have a million dollars in the bank. Engineering co-

founders are something that money can't buy.

Ask any technology manager or startup founder these days and you will hear some mix of these complaints about the "typical" engineering team (maybe not their own -- it's too dangerous to badmouth your people). Engineers:

- Hate meetings. E.g., the 'stand up meeting' or 'no meetings'.
- Hate task management. E.g., can't estimate.
- Hate schedules.
- Hate email. Don't reply to managers.
- Communicate poorly in general. Don't like to explain what's wrong, why something isn't moving.
- Argue about what end-users actually want.
- Continually demand more compensation, or else leave.
- Expect perks or frequently compare to others' perks, e.g., free lunches.
- Don't collaborate with each other.
- Like the technology more than the customer problem.

A scene in HBO's Silicon Valley perfectly illustrates the importance of culture. Two engineers, roomates working on a sound compression app, bet on whether one could identify an object the other touched using only his sense of smell. While one is exploring the room sniffing everything, it becomes apparent that the two had accidentally built the same DRM feature. Each had thought he was responsible for its development. As a result, their CEO gives them their first taste of project management.

What the boys had before the popular new method of "Scrum" was not culture, but waste. It's wasteful to run silly pranks like the one they do while they should be working, yes,

Michael R. Bloomberg's mayoral bullpen

but more importantly, it's wasteful for two engineers to both be working on the same project independently because there was no communication protocol in place.

A friend who works for one of the big New York banks was half-complaining to me recently. There's a new CTO at the bank, and he wants to make the bank more agile, "more like a startup". He's tearing out the offices, pushing everyone into a big open rows of workstations, changing the hierarchical meeting culture, eliminating the "phase-gate" step-by-step design and build and test and test and revise and release process.

In a way it resembles the statement Mayor Michael Bloomberg made when he walked into City Hall in 2001 and got rid of the traditional "West Wing office" setup that most government follows. He took a desk in the center of a huge room and arrayed 100 top managers all around him. Wide open workplace. Just like at the giant company he founded --

Bloomberg LP, the most successful New York tech startup ever.

Listening to my friend talk about his bank CTO's push to make Wall Street more like Silicon Valley, I reflected on whether his company setup looked like mine -- or any of the startups I've been helping.

We don't have that big bullpen at Knotable. We have something very, very different -- we have 4 people in New York who work in an environment that perhaps looks like that setup. But we have 56 people in 15 other countries, no other offices, perhaps 16 timezones when we are working.

We are seeing setups like this at other early stage companies -- teams of 1, 2, 3 engineers shipping interesting projects for very little money. Stirplate.io, Halo Neuroscience, Myndset, and many more.

There is this other way to do things. And in this five-part series, we're going to draw it out in detail: how to build an app, how to get your idea off the ground and out the door, without any free massages or unlimited Seamless or 20% time. It's called Code as Cards.

Chapter Two:
Hiring Through Firing

In Chapter One, we discussed the predominant philosophy of Code as Craft, espoused by elite start ups and engineering outfits around the world, and the problems inherent to it if 1) you have an idea but lack the technical knowledge to pull it off yourself, or 2) have an idea but lack the resources to attract and to hire the people to pull it off. Talent, the common parlance goes, is not cheap. But that's not necessarily true.

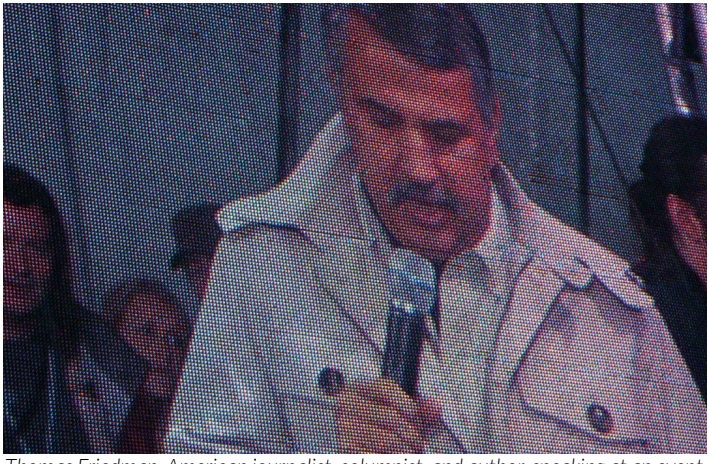
Thomas Friedman, in his book The World is Flat, makes the case that globalization has produced, especially in the technological industry, a kind of level playing field. The rate at which technical knowledge has spread throughout the world is incredible, and Friedman argues that this is a positive development insofar as it puts pressure on companies in the

Western world to adapt to that change.

Friedman's argument got huge global attention and it would be safe to call it the way most people think about globalization. We see the evidence everywhere. Knowledge-powered growth rates in China and India have made rich countries look pathetic over the last decade, and many of the key growth industries are driven by know-how in manufacturing, engineering, and especially software. Many of the largest new business created in India the last decade are software and digital knowledge outsourcers - Tata Consultancy Services, Wipro, Infosys, and many more.

So where is the evidence for this flat world in startup land?

Some people think they know why some engineering is still so hard to find. John Larson, an online programmer personality, wrote a popular blog post about this very topic entitled "Why



Thomas Friedman, American journalist, columnist, and author, speaking at an event.

I Will Never Feel Threatened by Programmers in India." In it, he defends the exceptional American tech talent pool, and recounts stories of Indian programmers who were paid $14 an hour, and consequently wrote code that needed, later, to be fixed by an American for a significantly higher rate.

Which obviously plays into a common cultural touchstone about outsourcing labor: you get what you pay for. Spaghetti code, poor architecture, hacks, shortcuts. The experiences Larson talks about have been widely felt among the Code As Craft engineers that run and build most impressive products today. Their institutional memory is large and uniform, and produces a knee-jerk "no way!" to outsourced networks of engineers.

Fortunately, the received wisdom is oversimplified and wrong, because there is a way to make outsourcing work. It just requires a very particular approach: you have to think deeply about who and how you decide to hire, how and how much you pay, and how to vet your hires.

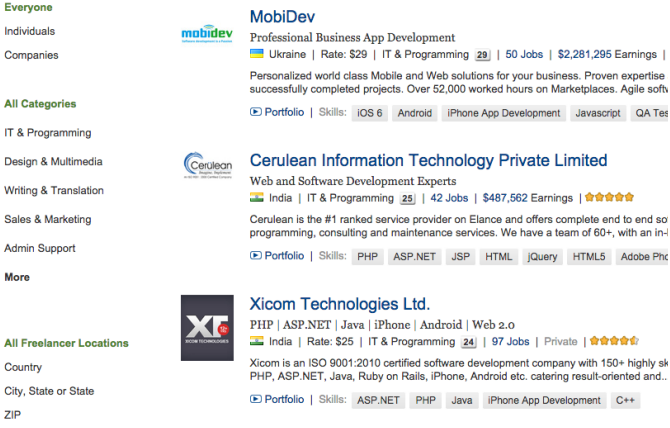
We call the people approach a "network" in Code As Cards. You need to build a freelancer network -- which takes work and maintenance. But this network can build quality code, quickly, cheaply, and it can be built quickly. When you have a network of freelance engineers, you have a flexible and interconnected team that can dial up and down to your budget and produce high-quality output.

Let's talk about how.

• • •

Finding skilled programmers has probably never been easier than it is today. With freelance hiring sites like Elance and Odesk and Freelancer.com, as a few examples, you can literally source entire world for programmers capable of doing exactly the kind of work that you need to have done, and even check out their reviews and ratings to see which ones are officially "up to snuff."

These are markets that look very different than your local "job listings". If your startup puts up a post on a tech job board somewhere, you are likely to get exactly zero candidates clicking into your job in the first hour or first day. If you do, you have to be waving huge compensation, full-time roles, equity, an exciting startup story with funding, and so forth. So for a start, with freelance hiring sites you are getting some replies no matter what you post. Great.



Screenshot of the Odesk marketplace.

Now the next problem. You have a lot of people to evaluate.

Maybe you want to quickly pick the top talent -- the highly rated folks. 5-star programmers on a site like Elance, though, will charge an accompanying premium, and that's just not sustainable for a cash-strapped operation. The top talent is actually good at working the system. They don't just charge a high hourly rate, they also bill projects on fixed cost budgets -- so they quote you something like $1,000 or $5,000 to do "the whole project". It's a bigger bit size. These operations are also busily surfing all the other offers coming into the platforms, so they usually have dedicated sales and marketing staff -- someone to write those friendly emails to you. This pads their cost as well. And more dangerously, sometimes these guys are just the sales guy, someone who in turn tries to cobble together some engineers behind the scenes.

Yikes. That shortcut won't work.

So there is a high ratio of noise to information with these folks. The natural solution? Scrutinize these folks. Dive into their portfolio, get references, stay up late one night and interview a bunch of folks via Skype. Maybe even have them do some detailed spec work in advance of awarding the project.

It's a solution to the Larson problem. Right? You want to hire coders from low-compensation countries like Ukraine, who are willing to work freelance on sketchily described projects, but since you will weed through the dozens of resumes and pick the stars...you'll avoid the pitfall of spaghetti code.

This isn't a shortcut, it's a longcut. But it won't work either. First and foremost, it suffers from the world's oldest problem

with hiring -- interviewing sucks. There is a long-standing body of research on how useless interviewing is, like this piece by the cognitive psychologist Daniel Willingham from University of Virginia in the Washington Post:

> *"You do end feeling as though you have a richer impression of the person than that gleaned from the stark facts on a resume. But there's no evidence that interviews prompt better decisions."*

That was the longcut -- to interview people, for 20 minutes, by Skype, in the middle of the night, in broken English. Of course, if it doesn't work for universities and large American companies locally, as in Willingham's research, then it probably won't work for you.

They do say the resumes are useful, but unfortunately the world of freelancers has very mingled and confusing profiles. You won't recognize anybody's schools or grades or concrete accomplishments from browsing through their profile. In fact, close scrutiny to profiles walks you into the second dangerous aspect of this longcut: your time.

Your time is precious. The ideas are yours. You have stuff you need to get done. If you dive deep into the land of evaluation and hiring, you are doing that. The more time you soak into a first hire, the more your investment increases, the harder it will be when we get to step two of the system (how to manage the work) and the more you will be disappointed when your first experiment fails.

We've worked with folks who tried Code As Cards and crashed and burned on this step. They "overhired" and got stuck with someone they couldn't move on from. Don't do it.

And anyway, the whole point of Code As Cards is to get you up and rolling fast.

There is a way.

The answer is so simple that it almost hurts: fire people as quickly as you hire them.

Code As Cards relies much more on firing than hiring.

So here's how you hire: post a job with very brief statement about what the project is, select a low compensation rate like $15/hour as the maximum, and specify a rule banning any "teams" or "companies", preferring solos, and specifying a skill or two. We usually post something like, "Hiring for a web app. Skills needed: Javascript. First 10 hour paid trial increasing up to 40 hours per week. Maximum $15/hr. Send your github and trello usernames. No teams."

Then we just hire 5-6 people who reply to this without scrutinizing them at all. We just say, "Hey, please send your usernames. We meant that!" and then, "You're hired. Here is a link to read."

With project management tools like Trello and Github, which will be covered further in the next chapter, the vetting process has never been easier. Say you hire five engineers (wherever they are -- foreign, domestic, skilled, not skilled) on a freelance basis on a Monday. You then invite each one to a password-protected blog post that outlines the project and assign them all a "card", or a small task, and say that they will be paid a certain amount for a couple hours of work on it.

It may sound silly to pay out $100 to a bunch of random, un-vetted coders, but what you're looking for is reliable,

quality workers, and in the end it's a small price to pay. Much cheaper than committing $5,000 to a team on their proposal or spending weeks interviewing tons of people. Spend $100 per person to find out if they are good.

Cap that initial exposure. You give each of the cards a short time-frame, to instill in your potential engineers the fact that this project is about having a quick turnaround on small, compartmentalized tasks. On the following day, you assess the results. If what's been produced doesn't meet your specifications exactly, fire them. "Thanks, but we don't need any more work." You may only end up with one viable engineer, but that is just fine: you've just hired that person for far less than you would have otherwise. And after forty weeks, you can have twenty people working for you for just $300 a week – people that you know can do the work.

This matches our experience and we've seen it in multiple companies. For every five you hire, you end up keeping maybe one. So keep doing it and you start having reliable people. And for every ten that you keep, you find that one is a star– a really extraordinarily motivated and high impact person.

At the core we are filtering on two things when we hire people: do you have the *will* to immediately start contributing, and do you have the *skill* to immediately contribute? If it's a match, there will be progress right out of the gate and we reward you with more work.

There are some small tweaks that need to be done to our ordinary idea of hiring, of course, if this is going to work. One key element is using strong frameworks to de-skill the

engineering. You want to make sure the cards are tiny tasks that still *produce visible work* – one major pitfall of remote working is failing to make sure that the work is actually happening. And finally, you want a high-visibility, live environment for those results. You want to lower the latency on feedback, but you also want your freelancers to be motivated by peer pressure.

In the next chapter, we'll really get into this part of it: the work. How to build a project, how to assign tasks, and the tools to use to get it all done.

Chapter Three: Visible Work

Last time, we talked about how to hire and fire, how to pay, and overall, how to assemble the kind of workforce that you need to make your idea for an app work on a tight budget. Now we need to address the tool set you need to make this system work.

Making an app from scratch ("as craft") is difficult and time-consuming when you're working in the same room as people that you know. When you start delegating duties to engineers all around the world, the problems get multiplied. That is, unless you have the right tools at your disposal.

These tools aren't that well known. But used correctly, they make the philosophy of Code as Cards possible. In the last chapter we touched on attitudes that experienced engineers

have today about turning to remote, freelance engineers. Those attitudes come from past experience and past tools. But things are different now; different enough that it can work. And it's very concrete. We'll lay out a few tools very concretely.

But first a big idea.

Let's begin with the concept of "visible work." You're not hiring "team members" or "colleagues," not really. You're not hiring someone who you feel like you might want to chat up at the water cooler. You're hiring someone who can do the work. That's obvious enough. But Code As Cards focuses intensely on the work. The hiring process completely ignores the "who". It just says "start working".

Visibility is the crucial added aspect. Engineers can give some pretty compelling reasons for why they have to do "invisible work," or work that you can't quantify, see, measure, evaluate. These are smart people. Research, planning, rebuilding, refactoring, cleaning up, organizing, simplifying, editing, improving: these are some keywords to think about. They aren't exclusively engineering concepts, and invisible doesn't mean they are bogus.

But a lot of them are in fact invisible. Let's do a non-technical example. You are reading this book. Maybe you are reading it to get better at managing product development projects. Worthwhile! But to any observer, there isn't a measurable difference on your desk or in your work output... yet. You are doing invisible work.

If you had a contract worker for your think tank working in a far off country, you would understandably be concerned that

this contractor was goofing off or perhaps not reading closely or somehow doing something useless. What's the solution to this risk of outsourcing the job of reading this book?

Option 1. Do it yourself. Or maybe hire some local person and have them sit in your office. Watch them work. See that they are doing it. You see the parallel to the way tech companies work here, no doubt.

Option 2. Ask the researcher to send a daily email with a synopsis including at least five main points from each chapter. After the first email you will see a) "is this person reading this stuff in a timely way or goofing off?", and b) "is this person turning the reading into something usable by me?" Here we have taken in invisible work and made it into visible work. If the researcher read a chapter, there should have been a short email showing what they did. Simple cause-effect.

As Peter Drucker, the management guru has written, if "you can't measure it, you can't improve it". And this is the operative concept for making work done out of sight into work you can easily observe, measure, and improve.

So, visible work. That's what you want. How do we make this relevant for software? After all, writing code is in fact visible. The engineers write some code. You can count lines of code they write. But good luck doing that. It's like saying you can measure many hours someone is at their desk to see if they are contributing at the office. It's visible, but it might not be work.

You could also just read the code, and see if it's good stuff. Good luck with that one too. You need nearly as much expertise

and time as the engineer in the first place. You replace the problem of coding with the problem of measuring.

Visible work has to be easily observed so you aren't spending tons of time figuring out what happened and when and how; and it has to be meaningful – so it moves your product agenda forward.

• • •

Now for the how-to.

First you need a place for the code to go. When someone writes some code and hits save, it needs to be somewhere you can see it.

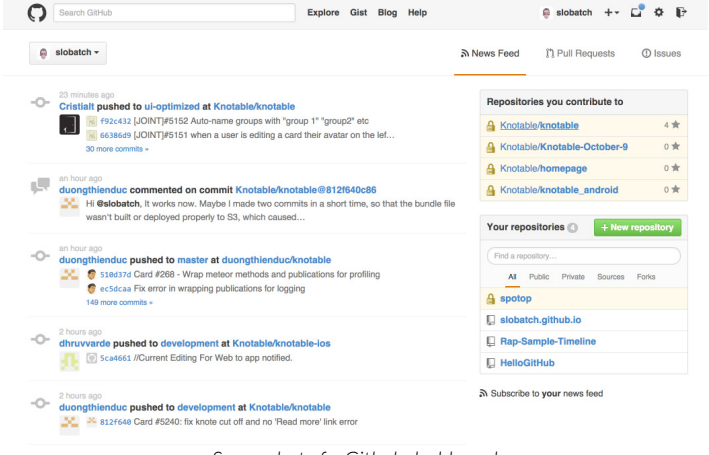
Github, the premier platform for collaborative code-writing, does a lot to assist in the process. Not only does it allow the entire group of engineers access to the same code, it also opens up an avenue for peer review, which is essential, especially when it comes to working with engineers from all across the world. You can eventually make quality assurance a task in itself with Github: you can have coders all checking each other, making concerns about the quality of these freelance networks basically moot. And with save-to-deploy, a coder in China can hit "save" on a card, and you will see the results immediately on your iPhone in NYC: now that's visible.

The change-control features in Github (and in other similar tools, like Bitbucket) help you manage other risks. You can see changes and revert back if something isn't good. You can see who changed what. You and all the engineers have access to all of the code, so they go find places where some interdependency is causing problems. Essentially, it gives you

a safety net for the incredible risk you take in giving near-strangers access to your entire project. What if some dummy deletes everything or inserts a virus? Just click one button.

Github is also your training ground and onboarding for new developers. The way the project code is organized and documented in this environment should include some "readme" files that say how to use it. You have your first engineers do this documentation from the outset. You ask someone to write a "New to this project? Here's how to get all the tools you need and begin" document, then you assign a new coder to actually do those steps. If they have problems, they complain and you ask them to edit the document for clarity. Iteratively, the team improves the readme material. And at the end, you no longer have to train new people. You just pay them to spend a little time reading the self-help documents.

There is just one thing you need to track in Github.



Screenshot of a Github dashboard.

Don't get lost reading people's code. Just track "commits". A commit is when someone saves their changes into the overall repository. Like a Facebook or Twitter feed, you can see who committed what and when, and you see a short line where they explain what they committed. That's all you have to track and understand most of the time. Because this visible work is going to be tracking closely to the next layer of the stack: Trello.

Trello is a project management app that lets you plot out whatever you're working on in term of cards. It borrows from the Japanese system of kanban (in Japanese, kan = "visual" and ban = "card"). Trello is based on the board, the list, and the card. Boards house your projects or products: the metaphorical whiteboard. Then there are lists, which give you columns with which to organize your projects into lists so you can understand it visually, in terms of tasks. And lest you protest that projects don't always flow linearly, that's the beauty of Trello: it lets you embrace complexity, and move things around, position them as parallel tasks, or on-going ones, or whatever else you need.

Lists, finally, are made up of cards. Cards are Trello's building blocks. With Code as Cards, each card that you create should be, to whit, the smallest observable increment of work you could possibly observe. This is difficult, and requires some discipline, but in the end will ensure that you're initiating productive work that can be quickly incorporated.

Instead of Trello you might choose a different project management tool. Asana is one, and Pivotal Tracker is another. There are many. Here are the keys, though, to making this a system for cards: a) it should be a group view where lots of

people can see what tasks are posted, and b) the tasks should be constrained and simple so they fit on little visible objects like cards.

By making a group view you save a lot of the updates, summaries, and "what's happening?" project information dissemination. You just don't need to do it. Someone expert on the project can see all the things people are working on, the things that are to do next, and the things that folks say they completed. Someone entirely new to the project can eyeball the overall picture, then pick one card and say "I'll try this one". A manager can glance at the lists and see if cards are moving from left to right fast enough, and hassle folks that are sitting on cards.

So just as you add someone to the Github repo when you first hire them, you add them to one of your project boards – maybe you start with just one but over time have separate boards for separate themes of work. Instantly they can see what's happening and get started. You assign them a card yourself or maybe they pick one to try.

Now here is where visible work comes back. Taking a card assignment is visible – their name goes on it. Then it needs to move visibly. From "to do" to "doing". And after a little while, maybe a day or two, to "done". When it's in "done", the engineer is saying it's complete. You should see a commit in github that corresponds to this – they should even name their commit with the name of the card they were assigned. You have a really clear way to see what someone is up to.

Here is where the second key ingredient of the project

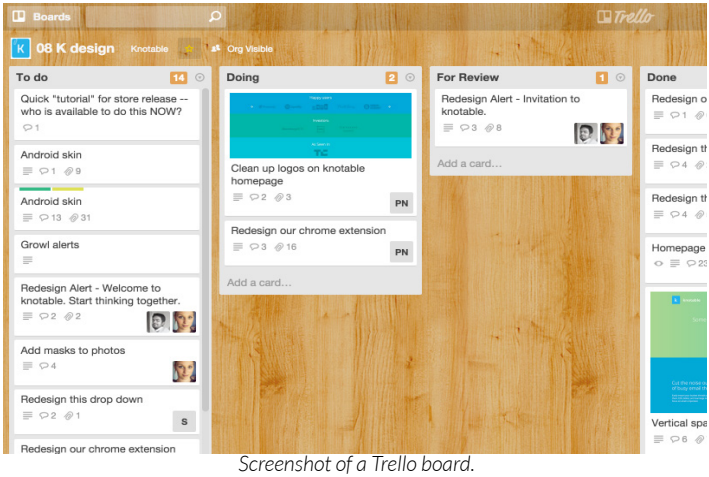
tool comes in. The cards have to be short and simple. A card is going to say something like:

Login button should be square-corners and turn blue when clicked.

Now that's a really simple little ask. It's easy to understand and it should be quick to do. If you assign someone the task of tweaking the appearance of a button a little bit, and they can't move it from "doing" to "done" pretty fast, well, you have a problem on your hands.

Small, simple cards are ideal for visible work because the cards have to move. If the card demands an output – is the button blue? - then there has to be a commit associated to. The code changes. (Cards that don't have an output are therefore dangerous. Don't do those.)

Let's dig in on simple a bit. Everyone comes to Code As



Screenshot of a Trello board.

Cards doubting whether everything is simple. Everyone thinks the blue login button is nice, but how do you explain the entirety of My Awesome App in those terms? So no need to fear. Let it out. Vent a little. It's a difficult step.

The principle at work is that every complicated idea can be disaggregated into a collection of small, visible tasks. We will explore this more in the next chapter. Writing cards effectively is really important. But for the time being we only need to believe that if you write short cards, the visible work system hums along.

If you write long cards, it dies. Engineers start saying they are "30% complete with a card" or "need another few days" or "it works on my machine but I need to refactor it to merge with the major codebase". When they say this you start losing the ability to evaluate skill and will. Is this person lazy or have they gotten distracted after doing lots of good stuff? Are they stuck on a hard area and unable to realize this? Even good people can "change" in this way – something comes up or they get assigned something too hard. If you can't track visible work at all times and just trust people at their word – people you barely know and never see working – then you are going on very little information. Don't do it.

Not only can this be a difficult road, you won't have everyone's support. Your engineers themselves might hate it, especially if your organization already has some clever, talented local engineers. But even among the freelance network you are building you will meet resistance -- the "model builders." Some people want to see the entire puzzle at once,

and when confronted with a small task may claim that it just can't be done, that you haven't thought through the project in its entirety. They'll debate you and they are smart. You may lose your nerve and back down "just this once".

To avoid this "antibodies-against-a-new-approach" response, you have to select your team carefully. The hiring approach in Chapter 2 is designed to weed out the "big thinkers". It's made for the hackers-at-heart, people who are willing to say, "I don't care about theories. I made the fix. Check it out please. Pay me." Pragmatic folks love this approach.

To resist the antibodies-against-a-new-approach response, in the case that you already have engineers on board, you are going to struggle with the following feeling: "But I don't have the knowledge required to make this work! I'm not an engineer!" There's something you can tell them: "You probably do know what you need to know. Just start with something small and visible and keep moving one step by one step." Folks can't argue with one tiny step of progress, they can only predict you will eventually fail. Carve off a small project somewhere to try Code As Cards, away from the antibodies.

Here's another problem you may have noticed by this point. Your developers have now committed some code and moved a card. You see movement. But what have they done really? Is it any good? Does it work? This is another step where people in the past got stymied. If you have to actually read their code, you will spend too much time evaluating. And in fact part of how we got here is that you aren't super technical yourself. If you could write the code yourself you would not be reading

this manual for shortcuts. So is there a better way to evaluate the work? The next level of the toolstack answers this question. Let's go there.

So far we have a code repo and a project board. The next place we will go is the app framework.

These days the tools and environments people build with make it really easy to go from code to something you can use. It used to be a long process of "releasing a test build". Now it's really simple. You can ask your team to glue together a few things so that the following happens:

1. Engineer works on some new code, testing it locally on their own computer. Then they hit "save" and commit it to the repo.
2. Once in the repo, a continuous integration tool notices the commit and packages the entire software project to deploy it to a test server. This can also be done as a quick command or two that the developer is asked to execute whenever they commit.
3. A virtual server like one from Amazon Web Services or Google App Engine gets this package and runs it.
4. You can visit this web server on your browser – on your smartphone even, and see the live code.

In our projects we have relied heavily on the new web framework called Meteor, which is a bundle of frontend and backend Javascript code that can do pretty much all the things various web applications and servers are designed to do. It has many packages and add ons that are easy to activate. It's free and open source, and ideal for getting your project going.

They'll probably have some business model that you can consider once your product is really big and humming.

Meteor comes with a hosting environment, so you don't really need a separate virtual server. Your coders can type a command after they commit code, and send that code to TestA.meteor.com, for instance. Then when they update their project cards, you can visit that server and see if the changes are there.

Meteor's credo:

> *"... [the] new way should be radically simple. It should make it possible to build a prototype in a day or two, and a real production app in a few weeks. It should make everyday things easy, even when those everyday things involve hundreds of servers, millions of users, and integration with dozens of other systems. It should be built on collaboration, specialization, and division of labor, and it should be accessible to the maximum number of people."*

Frameworks like Meteor basically make it so you can build an app without having to think at all about architecture. Architecture is one of those scary "invisible work" topics that can make Code As Cards difficult. But here inside the Meteor framework, you don't have to analyze and debate the merits of MySQL vs. MongoDB, those choices are made for you already by the framework. A bunch of stuff comes with Meteor, you use that and just focus on your own unique needs.

There are other frameworks like Meteor for the web, so use what you want. But the main point is – the new generation of developer frameworks free you from agonizing decisions or expensive specialist greybeards who tell you such-and-such operating system plus database cluster design is needed. (This

is true for the hardware side too – once upon a time you had to pick your servers and their setups. Now you rent them from someone, with their expertise on design and efficiency already layered in.)

Another critical benefit of using a strong framework like Meteor: it's idiotproof. As they loudly advertise on their marketing materials, Meteor is "all Javascript". Javascript happens to be the most widely known and used programming language. Any idiot can code Javascript. This means your hiring pool is massive. Sometimes techies get interested in trendy elite languages like Erlang or Go. They probably have their advantages. But the advantage of Javascript is that anyone can do it, and so the cost for recruiting and maintaining is low.

There is a reason that so many people learn Javascript. It's not just that it's popular. It's that it is easy. Javascript handles lots of problems that other languages made coders think hard about. Architecture, memory management, syntax. Javascript is really permissive on this stuff. It fixes problems for you. You can get away with worse code. Now the purist might say "hey, this is a cheap shortcut". For Code As Cards, this is just a plain old shortcut. Technology is doing the work. Think of it as a spellchecker, or the auto-save feature for Microsoft Word.

Meteor is a web framework. But a lot of these truths apply in mobile platforms for iOS and Android. The developer pool for Objective-C (the iOS language) and for Java (on Android) is smaller, admittedly. But they have done a lot to make coding faster and easier. The easy-to-test facts are still true. Any time a coder hits save, they can publish a build of the software to

you via a testing app like TestFlight or HockeyApp. You can get multiple versions of the app per day.

Using Trello, Github and Meteor in the ways outlined above basically ensure that even a novice product manager can put out a clean, easy-to-use product with minimal expenditure of time and money.

One Code As Cards-based startup put out an iPhone app in 2.5 months with out of pocket cost that surprised a roomful of practitioners. The startup founder managed the entire project on her own from start to finish. The app hit the App Store and has been getting initial feedback and traction. She had received quotes from ultra-lean, New York-based "minimum viable product" development shops for the project before she started. Pricetag: $150,000 and three months. Another venture-capital-backed CEO heard the story and said "Shipping that product for $50,000 would have been a steal." Her actual cost: $5,000.

The fourth layer of the stack is the highest up, and its job is to help you coordinate the overall work. You can dive in to the first few layers and produce results solo as we have laid out. But as you start managing more than 10 engineers, you will need some help and coordination – product management help. Roughly one manager is needed for every ten engineers. Questions come up, feedback is needed, cards need to be tested and refined. So working that pipeline of commits and cards and testing requires work and attention. And of course you have to feed the pipeline. With this big team working away, someone needs to think up all the tiny fixes and improvements

and new "cards" that need to be written. More on the work of product management in the next section.

In this section though, comes the question of visible work and coordination for the product management team, and the tool we have built and use for this at Knotable is... Knotable. Here is where we manage the discussion, prioritization and decisions of the product managers who are working on big bundles of product features or tasks.

A "knotepad" is a shared board or notepad where a few people can see, post, and edit "knotes". The knotes are messages or lists or notes like "These five things are my top priorities this week". Since everyone can edit them, they can be kept up to date. They work seamlessly with email – you get an update when there's a change, and can reply back with your comment – so it stays integrated into people's inboxes. The knotepads let you comment and annotate, so you can have a more complicated discussion than in the linear thread structure of email. And you can use more than words – post task lists, polls for making decisions, deadlines, files – so it has some simple project management aspects that make it more functional than just writing a weekly email. In fact, using Knotable in this context largely replaces the "weekly meeting" or "can we meet and discuss this change we want to make?"

So while engineers are making commits that reflect cards moving and can be tested in your phones web browser, the product managers are updating their knotepads to show where the progress is and even working through decisions through back-and-forth discussion with you.

As with the engineers, you will meet business people and product managers who "just want to get in a room and discuss" or who repeatedly turn back to email threads. But the benefits of the toolset are important.

Canceling all meetings from the workflow means you never have to be in the same place at the same time. You and your people are more free. You can work perfectly asynchronously. You can get the best people and get the best parts of people (maybe they only want to work evenings?).

Canceling all meetings also means that everything you discuss and decide is visible work. There is a pad with a task list or a chain of comments analyzing a decision. You can bring someone into the "discussion" after it has happened and get their full weigh in.

You can do something you might have had a meeting about "slowly" – brainstorming ideas for names over the course of time or collecting research on a new feature. You can let people get into "flow", so they have big blocks of time to get into detailed analysis and planning of the next key feature. Testing complicated scenarios in their products.

We all know meetings suck. These are just of a few of the reasons why. And with knotepads, you can avoid them. And on the other side you can avoid the other universal office pest – email. Emails get fragmented, lost, go out of date by the time you read them, fail to move the conversation forward, and all the rest. A way to focus work without email is what you get with a knotepad.

Now you have the last layer of the stack – a way to make

knowledge work like planning and designing features into visible work, and to extend the effectiveness of Code As Cards' global networks of freelance engineers to the product management function itself.

In the next chapter, we'll go over what comes next. You've hired engineers, you've gotten the toolstack together, and you are about to start assigning some cards. Time to figure out what those cards and pads should say, and what your product managers should be writing.

Chapter Four:
Chapter Four Title

So you've come this far -- you've got an idea, and you've gone through the necessary channels to find the right people for the job. You've hired programmers through Elance, set up Github and Trello, and gotten started on the project. Homestretch, right?

Well, somewhat. Software development, even using these techniques, is not something that runs on autopilot. Even with frameworks like Meteor that let novice product managers coordinate an app's development, there is a (relatively small) amount of active participation required on your part -- or on the part of the manager you've hired!

In the last chapter, we talked about visible work. That's what you're hiring these engineers for. But even with visible

work, there's some maintenance required on the part of management to make sure that that work stays visible. What do I mean by that? It's easiest to think of visibility in four layers.

The first layer is the engineers and contributors who have signed on to your project. The second is what happens on the level of apps like Github: like the "Track Changes" feature on word processors, Github records each instance of an engineer taking a piece of code and changing it. You don't quite know exactly what kind of a change was made, but you know something's happening, you know that an engineer has made a commit. The third layer is project management apps like Trello, where you can contextualize those changes in terms of the larger project, and the fourth is the live working output of the actual code itself. Working in a framework like Meteor is critical here. With this fourth layer, the first three come together into something tangible, something that you can actually look at and measure to see if its successful: live, committed code.

In Code as Cards, project management is all about making sure this evolution, from the first to the fourth layer, goes smoothly. The developers at Tint made a blog post last year explaining their approach to management. They found that they had some issues using the kanban system, previously discussed in Chapter 3, because members of the team were at odds when it came to knowing who was responsible for moving cards around on their Trello board.

The Tint developers also say that they optimized their use of Trello by ensuring that each card represented a small action

item. This is essential: cards cannot represent anything more significant than a single developer can finish in a small amount of time. If too many items are incorporated into a single card, you open up a space for your engineers to perform invisible work. Keeping things small and simple ensures that tasks get done one after another after another.

Think about it this way: if you set up a check-list for everything you need to do in the morning before you go to work, how are you going to set it up? You could tell yourself to make breakfast, get dressed, clean up, and get ready for work. Or you could tell yourself to eat a cup of Greek yogurt, put on your jacket and pants, shower for fifteen minutes, and pack your suitcase. One of these methods introduces ambiguity; one of them cuts down on it as much as possible.

But this is to go back to Tint's first point, about the kanban system being ambiguous for them. This is a problem of project management that can be solved quite easily: just give one single person the authority and responsibility to check in on each phase of the project and move cards as they need to. Simple as that! Cut down. Of course, keeping cards small is a challenge in itself. This is because the idea of Trello inherently involves compartmentalizing your project into many small pieces. What ends up requiring discipline is enforcing a mentality of moving just the cards, which means that your cards need to, in their totality, be the project.

Now, things do get a little complicated once you get past ten or twelve engineers working under you. This is where what I said earlier about hiring product managers comes into play.

Ten or twelve is really the maximum number of engineers that can be handled by a single manager. Beyond that, you need to hire people to oversee the visible workflow, people to follow up on cards and make sure they move, people to write good cards as things come up, and people who can delegate duties.

For proof of this, one only need look at a comparison of retail giants Staples and Amazon, as discussed in a recent issue of the Harvard Business Review. Staples, the way Andy Singleton puts it, has large teams build software enhancements that get rolled out every six weeks, which are then sent to another team to be tested for three weeks, in what IT professionals might call "best practice."

Amazon, on the other hand, has split up their entire infrastructure into thousands of compartmentalized services. Teams handle only a handful of services at a time, and once they're done, they release them. A change is released out of Amazon about once every eleven seconds, which according to Singleton means that Amazon makes 300,000 changes for every time Staples rolls out a new release.

This is the epitome of Agile development, and it is the way that a software entrepreneur needs to think in order to successfully develop a new app. Forming large, clunky teams and passing massive projects around produces "slow and steady" results, whereas the Amazon approach produces a lot of changes very quickly. If you're wondering which approach works better, remember which company is Amazon and which one is Staples.

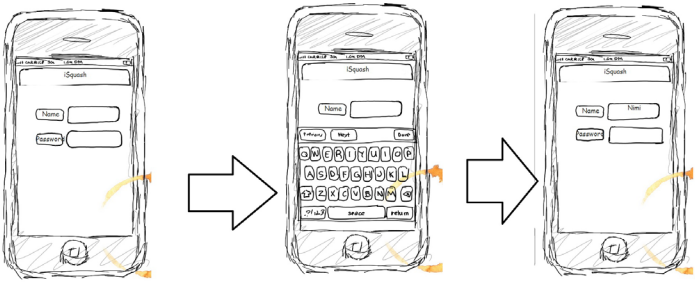
This all points to one of the essential truths about Code as Cards: a lot of this ideology is already at play in some of the major players involved in developing software. Another Harvard Business Review article heralds the arrival of a new theory of product development most prominently seen at Google: "To manage new software releases at their huge scale, Google has replaced traditional testing systems that depend on people with a testing machine, known as a "continuous integration" system... Continuous integration and automated testing is important for all modern, large-scale software development." We would add only that it's as, if not more important for small-scale software development.

The beauty of frameworks like Meteor and apps like Github and Trello is that they allow for this kind of rapid production to happen very easily: like we've said before, the fact that a coder in China can hit "Save" and you can see the results immediately on an iPhone in New York means that you can continuously assess the visible work your engineers are putting out, that you can constantly test, and that you don't have to waste time with bulletproof schedules and lengthy periods of gestation.

In Chapter 3 we talked about the kinds of cards to assign – a log-in needing to be blue with square corners, for instance. This is product development with small, disaggregated cards. Imagine a storyboard for a mobile app.

Here's the key: assign one task. The first one that would produce valuable results. And then assign nine more. What you need, more than anything at this stage, is momentum.

Make a homepage with a logo. Then make it blue; put a sign-up button; create a form that follows; have an e-mail sent when that form is filled out; build a database to save sign-ups; add a login button on the homepage. Virtually every app is going to need something like this, so start there.



A mobile app "storyboard."

We brought up Google a moment ago: let's look at Google for a second.

The beginning stages of Google, obviously, would be pretty straightforward, at least from a user-experience perspective. But one of the primary factors in Google's success has always been kind of an abstract thing, and that's what happens in between you clicking the "Google Search" button or hitting Enter and the page of search results you get afterwards: speed. How can you assign "speed" as a task to your freelance network? Quite simply, actually.

It goes back to our Peter Drucker quote from Chapter 3:

measure and improve. Start off by being specific: what's too slow about the app? What screen? All of them, the first one? The login page? Here's how you fix it: tell a developer to put an onscreen alert in as soon as the server is hit. Then install a timestamp. Then have the developer add another timestamp once the page fully loads, and show the full time elapsed. Now you can assign a rather straightforward task: lower that time. That's just an example of how to turn an idea like "speed" into a step-by-step process.

This is also an example of where your Crafty engineers may throw up their hands in exasperation. "No, this won't work unless it's done exactly this way," they may say. Its somewhat similar to an old argument about wings and evolution. What good is half a wing, the creationist asks. Evolution could never produce a wing straightaway, so it must be God that created them! But it's not quite like that: wings didn't start as flying implements. They began as tiny growths, eventually becoming delicate protrusions of feathers that helped keep the animals that had them at a stable temperature. And further down the line, there were so many feathers that animals started jumping a bit better, even gliding a little bit, to catch flies, for instance. And eventually you get to fully-winged animals -- all in a step-by-step process. This is product strategy. Get momentum -- build something interesting and useful right away. Get your freelance network working and showing that they are useful, and start producing output that you can show other people. Get daily updates to the app that you can try out on your own. This is how products progress.