How NOT to Build a Service…

Mike Perham
Software Developer, FiveRuns
http://mikeperham.com

Hi, my name is Mike Perham. I'm an engineer at FiveRuns and today I want to talk to you about how not to build a service.

Who am I?

- J2EE/Java 1999-2007
- Ruby 2005-Now
- Author of the DataFabric gem - sharding for ActiveRecord
- My 6th startup since 1999

I've been working with software professionally for a decade now, mostly in Java. I started playing with Ruby a few years ago and recently started working with Ruby professionally. I've been doing open source for quite a while now, my first open source project was written in C using the Win32 API. I've since worked as a member of the Apache project on Maven, the Java build and project management system.

FiveRuns is my 6th startup - in the last decade I've had three fail and two succeed. In this talk, I'll examine what FiveRuns has done wrong over the last two years. If you want to see what we've done right, just go to the press release area of our website.

Who is FiveRuns?

- Austin, TX startup
- Focused on Rails monitoring and performance
- Install: pre-built AMR stack (FREE)
- TuneUp: community site for Rails performance tuning (FREE)
- Manage 2.0: LAMR monitoring service (subscription)

FiveRuns is a venture-backed startup in Austin TX. We're currently focused on the Ruby on Rails market, provide three products currently. Install is an Apache, Mysql and Ruby package which provides an quick way to start using Ruby and Rails with your own projects. TuneUp is a development tool and community, designed to help developers track down slow parts of their Rails pages. Finally Manage is our flagship service, a monitoring service for the full Rails production stack, it will monitor Linux, Apache, Mysql, Rails, mongrel and several other common components of a Rails application.

What not to expect

- Ruby
- Web services

I want to set expectations before I get too much further into this talk. I'm not going to go deep technically. You won't see any Ruby code nor will I be talking about web services, SOAP, WSDL or any other acronym. I hope you aren't too disappointed.

One last caveat: I've been working at FiveRuns for about a year now. Three months before I joined, the entire company was restarted. So I was not around during the design and development of the original Manage service. Since I was not a member of the original development team, there is a fair amount of hindsight and speculation.

Agenda

- On Failure
- Mistakes
- What did we Learn?
- Q&A

I've split this talk into three sections. In the first part, I'll talk about failure and what it is. The second part will discuss some of the mistakes FiveRuns made over the last 2 years in building Manage 1.0 and what we did or are doing to correct those mistakes. Finally I'll wrap up and try to determine a few things we can all learn from these mistakes.

During much of this talk I use the terms decision and mistake interchangeably. To me, a decision can result in a mistake. A big decision results in a big mistake.

Failure

- Why do startups fail?
- Failure = Mistake 1 + Mistake 2 + Mistake 3 + ...

Failure might mean different things to different people. Apple's Newton, New Coke, Nokia's N*Gage device. All were big projects with multimillion dollar budgets which are all considered failures today. But why did they fail? I don't want to get into specifics but in general, they failed due to a set of mistakes: people made bad decisions which pushed the project closer to failure each time.

We can think of project failure in these terms: failure is reached once the cost of bad decisions adds up to the amount of money dedicated to the project. Each decision resulted in a mistake and each mistake has a cost to fix.

Failure

- More ways to Fail than Succeed

We have to make hundreds of decisions before a project succeeds and some of these decisions can have very broad consequences. So there are many chances to make bad decisions that might sink a project. How can we reduce our chances of failure?

Failure

- Experience == More data
- Smart == Quicker to gather data?
- Money == More Headroom to absorb Mistakes

You can approach that question from two angles: how can I avoid mistakes in the first place? and how can I get a buffer to correct any mistakes I will make? Both cost money.

There's only one rule in software: hire the best people you can. You hire experience and hire smarts. Experience means previous work in the market and technology your company is focused on, and people who've made mistakes before. Ideally that experience means they won't make the same mistake twice. Smart, experienced developers are usually happily employed elsewhere. Good luck with your recruiting effort. While I'm focused on developers for this talk, this rule holds for basically any role in a software company.

Money gives you extra time: you can afford pay salaries while correcting mistakes. There's really nothing that can buy time except money.

Failure

- So how do we avoid Failure?
- Every mistake has a cost.
- Avoid those costly mistakes.
- Have enough money to correct mistakes.
- The remaining mistakes won't add up to Failure.

How do we avoid failure? Generally, priorize your decisions based on their cost and make sure those decisions are made correctly. How do we know we've made the right decision? We'll talk about that at the end.

Types of Mistakes

- Business
- Social
- Technical
- Really, really dumb

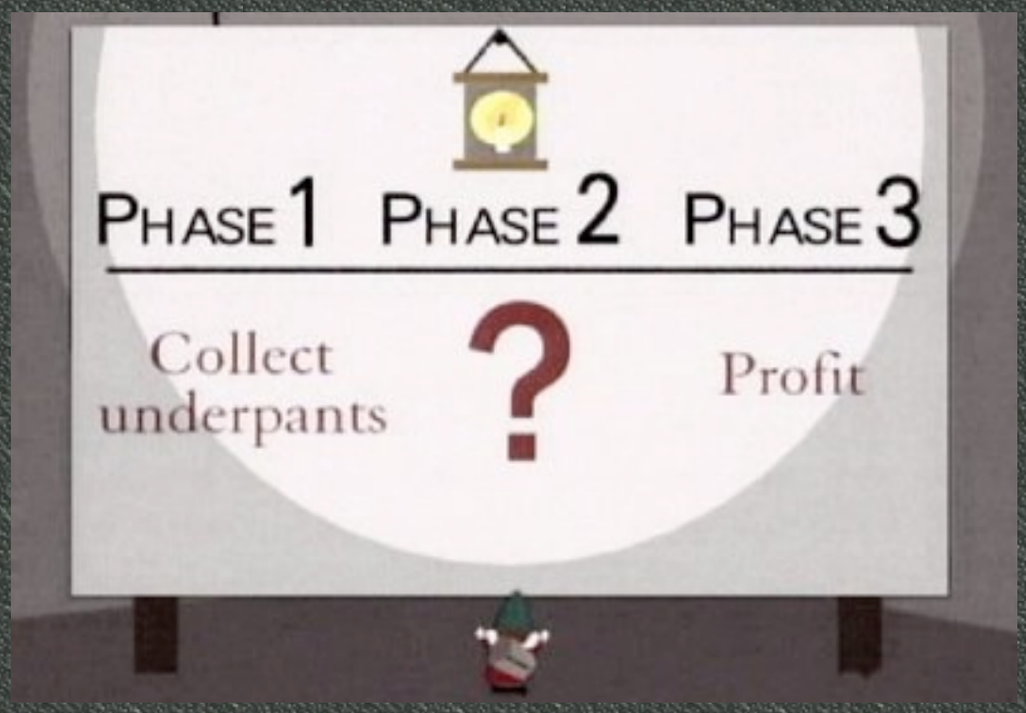
So mistakes are common and we want to avoid big mistakes: that much we can agree on. What types of decisions/mistakes do we make? Thinking about types of decisions can be helpful as it can help you decide on the size of the decision. In my opinion, technical decisions tend to be smaller and easy to correct if a mistake is made. Social and business decisions tend to be larger because multiple people need to become involved to correct any resulting mistake.

Business Mistakes

I'm talking about business mistakes first because in my opinion they are the most deadly. These business decisions tend to be core to your comapny's value and direction. Unless everyone working together has a common understanding of what they are trying to achieve, it's very easy for a person or even an entire group to work to no end or even work at odds with the intended business direction.

Know the Plan

- Who is your Customer?
- What are you building?



The overall business plan is the most important thing for everyone to know and agree on. When you hear about raising venture capital, the most frequent thing mentioned is the company's business plan. This is why that business plan is so important: it will shape the direction and focus on the company for years to come.

When FiveRuns was first started, we were a systems management company that just decided to build the service using Ruby and Rails. Our target market was enterprise IT departments and the hundreds of servers each had to manage. A year later, we shifted our target market to Ruby on Rails developers, selling Rails performance and monitoring service. Both markets are viable but by changing our strategy and shifting from one to another, we burdened ourselves with a big cost: we would have to rework much of our software so it made sense and provided value to the new audience. The needs of a few Ruby developers are very different from IT departments with 10-100 people. You'll see the effects of this change in direction throughout the rest of this talk.

Know your Customer

- What value will they get?
- How will you fit into their life?
  - Support?
  - Pricing?
  - Marketing?

Once you have a business plan with an audience or target market, you need to define your product's value. What does your product do and why will people be willing to pay for the product? These questions may seem obvious or simplistic but the devil's in the details.

Because FiveRuns started its life focused on the enterprise market, retargeting our service at the Ruby on Rails market meant that the product didn't deliver as much value any more. It also meant we needed to change or tweak many other things related to the service. Think about the enterprise market: we focused a lot of marketing on industry magazines and press releases, added "enterprise-ready" features and hired sales people. If you are targeting the enterprise market, that's simply what you do according to conventional wisdom.

Well, these things might make sense when targeting the enterprise but make little sense in the RoR world. Blogging, open source and conferences get us much closer to our customers.

Build a Good Trial

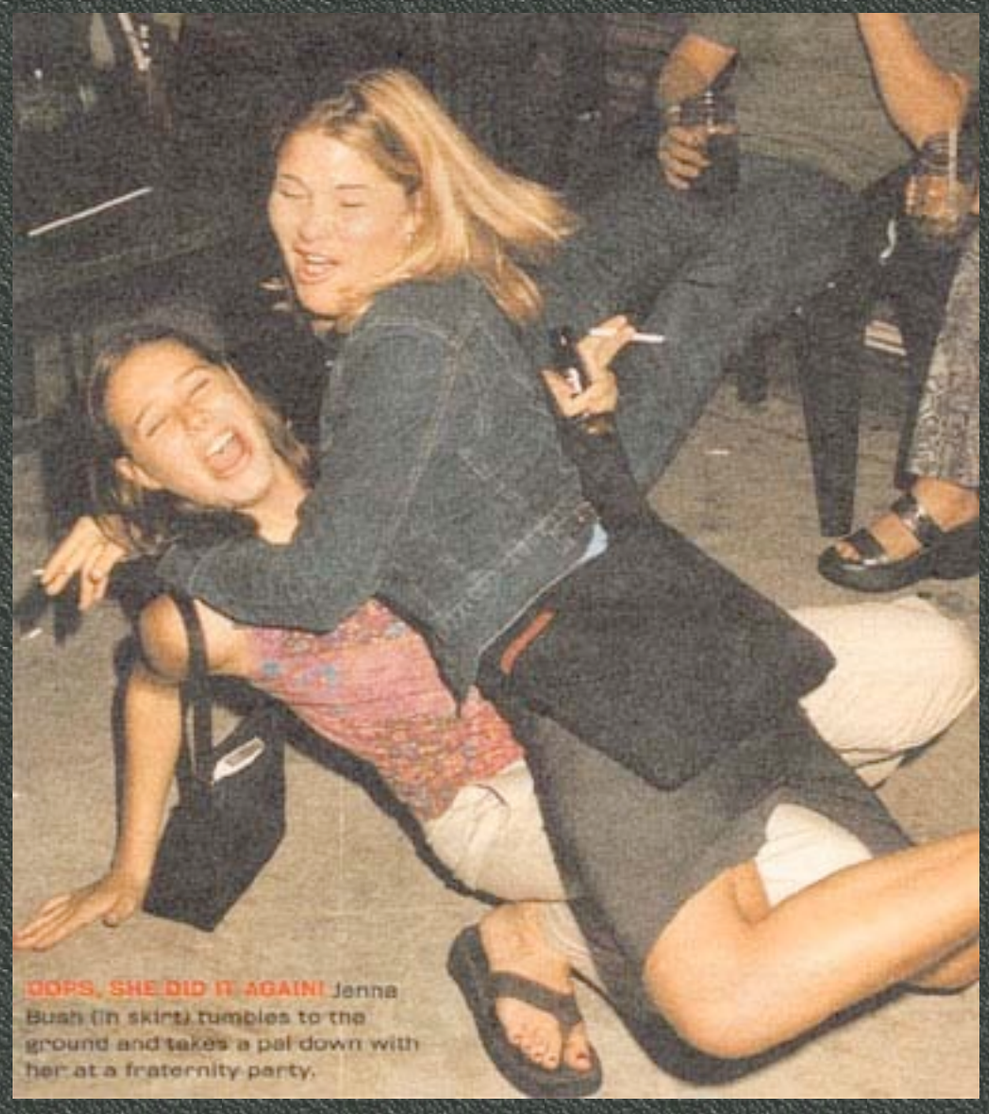
- Target Audience > Trials > Customer
- First part: Marketing
- Second part: Development
- It's all about showing value!

Once you've defined your audience and understand how to market to them, you need to convert them into customers. In the Rails world, this means a Product or Service Trial. Everyone offers a free level of their service or they allow people to try the service for a limited time. In our case, we offer a 30 day trial for our Manage service. The key to a successful trial is converting as many of those free trials into paying customers, this is your conversion ratio. For every 100 people who try your product, 70 might sign up as customers. This is a 70% conversion ratio.

In e-commerce, there's a concept known as the Product Funnel. This is the process from seeing a product on a website to purchasing the product. The customer has looked at the product so there must be some amount of interest in that product. The e-commerce site's job is to make the process of learning about that product and deciding to buy the product as easy and efficient as possible. Look at Amazon.com's product page: they show pictures, a description, reviews of the product, all in an effort to make you decide "I need to buy this".

At FiveRuns, our Manage trial is no different - the client software must be easy to download and install, easy to use and show value as quickly as possible so that the user can quickly decide that this service is worth paying for. Every bug or usability issue will result in lost customers.

Social Mistakes


OOPS, SHE DID IT AGAIN! Jenna Bush (in skirt) tumbles to the ground and takes a pal down with her at a fraternity party.

A social problem is one that involves people and process. I think of social mistakes as those which affect a group or department within a company.

No Ruby Experience

- Led to re-inventing the wheel
- Misuse of technology
- Non-idiomatic Java-esque code

Another major social mistake was using a technology we were unfamiliar with. Not a single developer had worked professionally with Ruby and since we were focused initially on the enterprise market, there was no real need to build the service in Ruby. I think a combination of optimism and marketing hype led the team to think that Ruby would be easy.

But Ruby isn't any different from any other technology. Without experience we're bound to make plenty of rookie mistakes. That's how we learn. And we made mistakes:

Capistrano is the Ruby standard tool for deployment. We created a bash script which deployed the entire platform. It wasn't flexible, it wasn't modular, but it worked. We spent more time than we should have reinventing a tool which already existed.

Our processing daemons which had no web interface still ran the full Rails stack. Why? Because the team didn't know how to run ActiveRecord by itself, outside of Rails or how to spin off a Ruby daemon.

The codebase itself rarely took advantage of any Ruby-specific language features, like metaprogramming and blocks. The previous team's background was mostly Java and they didn't have anyone to help them write good, idiomatic Ruby, so they wrote Ruby code which looked like Java. getters and setters, needless design patterns, we even wrote a custom JSON parser that ran 10x slower than the standard Ruby parser. Sometimes you need to take a step back and ask yourself "Why am I writing this code?"

So now we've hired Rubyists with years of experience to help guide the team in the Ruby Way. Specifically Bruce Williams and Adam Keys provide valuable mentorship and experience in guiding those of us with less Ruby experience in how things are done the Ruby Way.

No testing!

- No unit, functional, or integration tests
- No QA team or automated browser tests
- Very difficult to refactor!

There's not much to say here. Testing is something every developer learns to be critical to the stability of any system. Ruby doesn't even give you the benefit of a compile step. How an experienced team of developers wrote an entire system without tests is beyond my comprehension. Don't do this.

Our current test status is much better. Unit and functional testing are usually pretty thorough but in-browser testing remains lacking. To be honest though, I have yet to work at a startup that used automated browser testing.

Eating your own Dogfood

- "Here's how to monitor your systems"
- Our systems didn't use our service
- But how can a service monitor itself?

Finally, "Dogfooding" is a popular concept in software - it's using your own software. If you are a developer and you are building development tools, you should use those tools. That usage provides constant feedback as to the usability and stability of your product. Without it, your service is more likely to be buggy or have usability issues.

We didn't use our own software to monitor our systems. Now there is a bit of catch-22 here: how can a service monitor itself? Well, there will certainly be a limit to what it can or can't do but that shouldn't have prevented us from doing what we could. At the very least, we should have been using Manage to monitor our own laptops. Our latest products, including FiveRuns TuneUp, are used by us frequently and as a result, the community response has been very positive.

Technical Mistakes

Technically there were some architecture and design mistakes we made while building the Manage service itself. Notice that several of these issues are directly related to the change in market focus. When we moved from an enterprise focus to a Rails focus, there was plenty of design decisions that no longer made sense.

Java/C client

- Old client? Java/C native binary
- New client? Ruby & `cat /proc/stat`
- Before: 68,000 lines of Java + C + XML
- After: 5500 lines of Ruby (~92% reduction)
- Permissions problems remain!

The Manage service requires you to install a custom client on every machine, which collects runtime metrics about the various pieces runnng on the local machine. Our original client was written in Java, with a few custom JNI libraries written in C to collect various metrics from the operating system. I'm not positive why the client was originally written in Java but I suspect that IP and source code protection was a concern. Since Ruby is difficult to obfuscate without deep knowledge, the team used what they knew would work.
As for the C parts, I have no idea why they didn't just read the /proc filesystem or something as simple as that. I hope most developers would agree that C is overkill for this task.
When redesigning Manage, we decided to throw out the entire client and rewrite it from scratch. The new client is about 90% smaller in terms of lines of code and has a memory footprint about 60% smaller. It's a standalone, statically built Ruby binary.

Unnecessary Features

- Custom HTTP Encryption
- Custom Proxy
- Support for Jboss, Tomcat, FreeBSD, Oracle, Windows

When we were initially focused on the enterprise market, we built some features that turned out to be useless in the Rails market.

I don't know why but the old client didn't use SSL. Instead it used HTTP and encrypted the payload by hand. I guess they thought that the SSL port could be blocked at the corporate firewall but port 80 would be open. All I know is that I've never heard of a corporation where SSL is blocked.

We also built a pretty complex proxy functionality into the client in case the machines did not have direct access to the Internet. Instead one client installed on a machine in the corporate DMZ could act as the conduit to our servers.

Now note that we charge $40 per server per month for our service. For that low price, we can't afford to support the odd paranoid customer with draconian firewall rules who only allows HTTP via proxy. The new client talks SSL directly to our servers and we haven't heard a word of complaint about it so far.

Lastly, the Manage service monitored several subsystems which aren't really in demand in the Rails world. Jboss, Tomcat, FreeBSD, Oracle, Windows - all of these have usage in the low single digits in the Rails world. While we do see occasional requests for these, it's simply not cost effective for us to support them until they become more popular.

No Integrated Billing

- Before: Fax us your credit card details!
- After: Online purchase

Continuing with our e-commerce analogy, once the user has decided to buy your product, the checkout process must be as simple as possible. When you look at a product description page on Amazon.com, it provides a ton of data and links to make browsing and learning about products simple. But the minute you press the checkout button, the page because much simpler. The entire checkout process is designed to make purchase as simple and quick as possible.

Unfortunately our checkout process was horrible: we didn't have one. To buy our product, you had to fax/email/skype your credit card information and we would charge it by hand every month. This is something you expect from Mom & Pop Co, but not a technology startup. I don't know about you but if someone wants me to call or fax them in order to buy a product, I think real hard about whether I really want to buy that product.

Happily, today you can buy our service online without any manual intervention on our part. Yay!

So What can we Learn?

Mistakes in Software

- Recognize the Decision
- Prioritize the Decision
- More knowledge = greater chance your decision is correct
- Which Level of Ignorance can you afford?

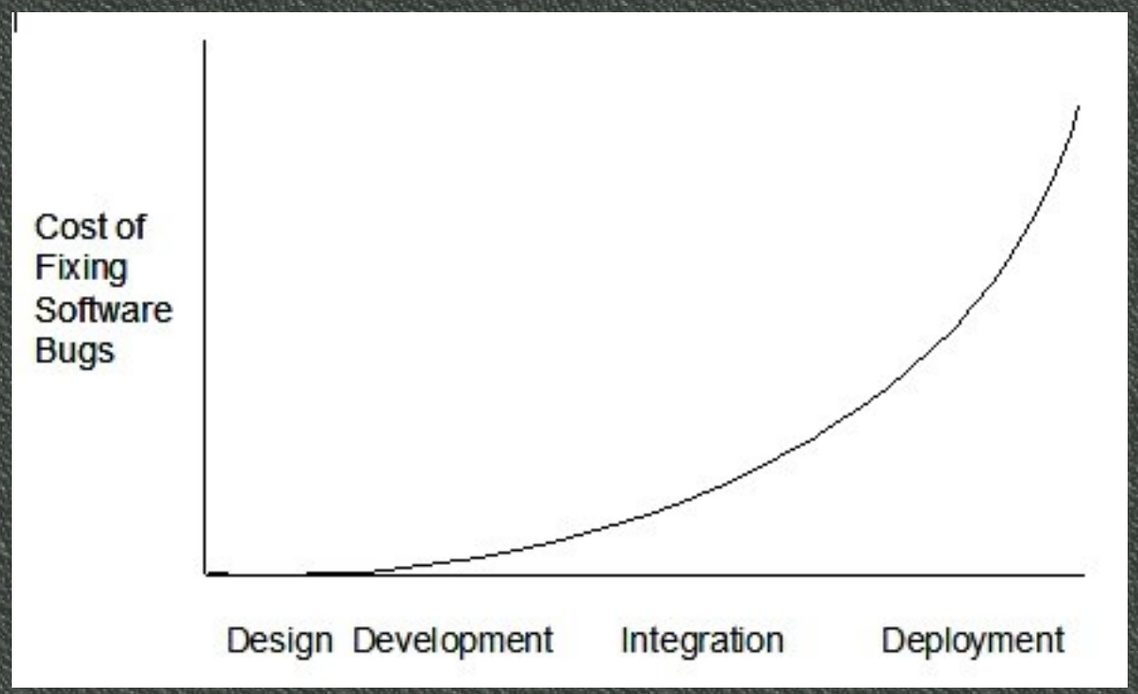
Everyone in this room knows that mistakes are endemic to software development. Every bug, every missed deadline, points to a mistake made by you or someone on the team. Behind every mistake is a decision and you make decisions based on some amount of data. As we get experience as developers, we unconsciously rank decisions based on their scope or size because the cost of making the wrong decision can vary greatly based on the decision we are making. We gather as much data as possible as insurance against making the wrong decision.

Levels of Ignorance

- 0th Level -- Lack of Ignorance (I know something and can demonstrably prove it)
- 1st Level -- Lack of Knowledge (I don't know something, but I know that I don't know it)
- 2nd Level -- Lack of Awareness (I don't know something, and I don't know that I don't know it)
- 3rd Level -- Lack of Process (I don't know something, I don't know that I don't know it, and I don't know how to find out that I don't know it)
- 4th Level -- Meta Ignorance (I don't know that there are these levels of ignorance)
  - *Philip G. Armour, October 2000, Communications of the ACM*

Take out Insurance

- Lower levels of Ignorance require more time up-front, of course...

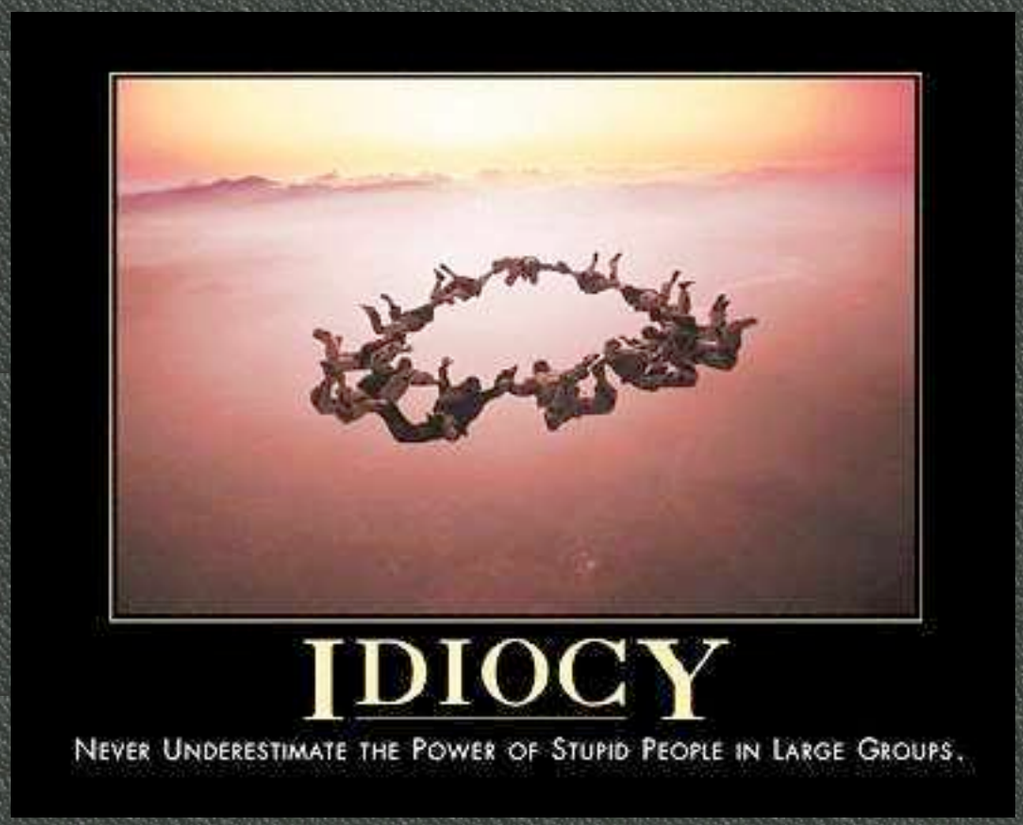


The best insurance against a bad decision is more data. You need to verify that your decision is correct. When making a time estimate for a task, that may mean that you take the time to list out the subtasks required for that task and estimate each subtask. You'll often find that an optimistic estimate at first glance leads to a larger estimate once the task is thought through.

If you are decided on the network architecture for your service, you might spend 2 weeks with a group of people thinking through all the requirements of your service to ensure the proposed architecture is correct. The reasoning is unstated but obvious: the larger the potential cost of a bad decision, the more money/time you spend verifying it is correct before implementing it.

Groupthink

- Groupthink
  - "groups make hasty, irrational decisions, where individual doubts are set aside, for fear of upsetting the group's balance" - Wikipedia

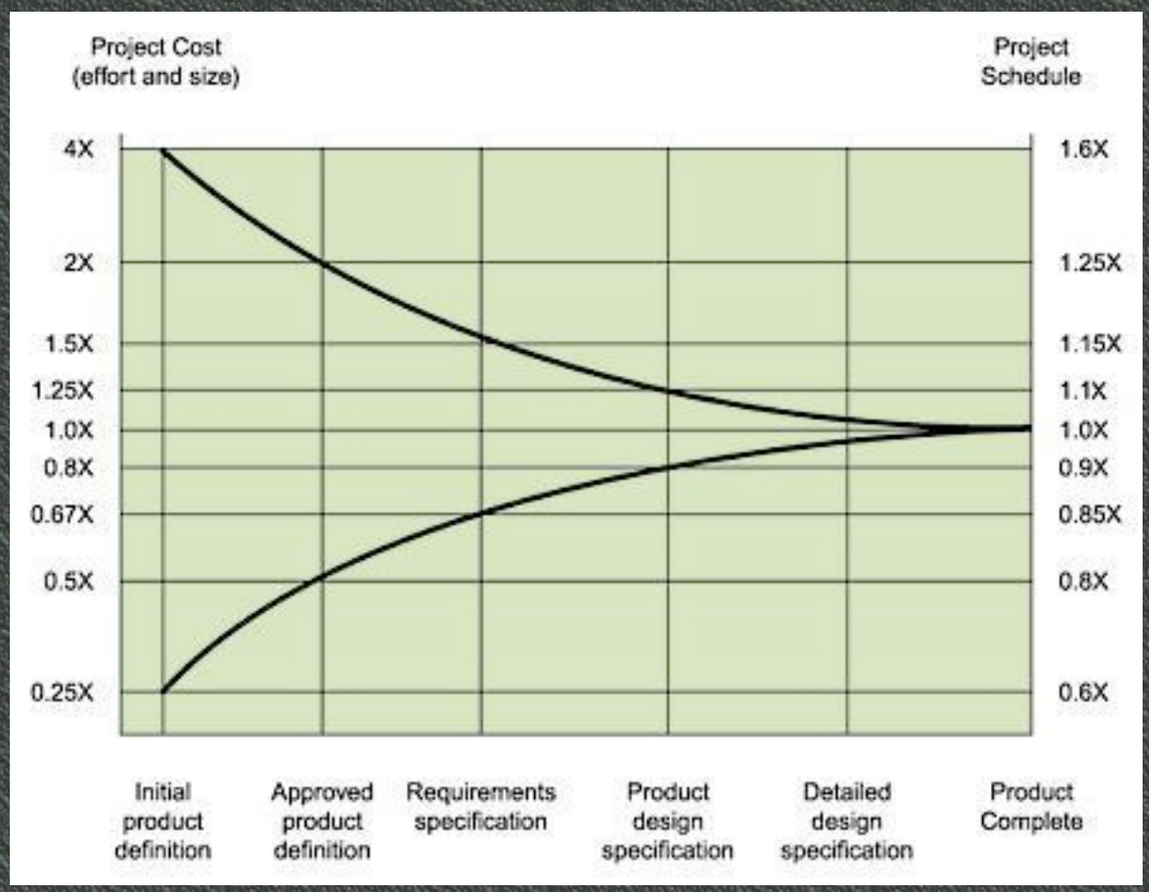


So once you recognize you are making a big decision with long-term ramifications, be cognizant of some common pitfalls: group think and optimism. You don't want to be a roadblock to progress but a little devil's advocacy can help and should be see as a necessary part of big decisions.

This commonly happens when you have an "expert" in the group whose opinion is not challenged. Times change, what worked in one situation may not be applicable today.

Hubris/Optimism

- "the first false assumption that underlies [...] programming is that all will go well, i.e., that each task will take only as long as it "ought" to take." - Fred Brooks, *The Mythical Man-Month*



What We Learned

- Changing markets = expensive
- Trial experience = first impression
- Use your own software!

As for specific things we learned:

1) changing your target audience is very expensive. It will affect almost every other decision you've made. If change is needed, you'll need to rethink your software's architecture and feature set.
2) the trial is important to get right. Your trial is the customer's first impression of your software and a bad first impression is tough to overcome.
3) dogfooding is vital to providing maximum value to the customer.

Conclusion

- Bigger decisions justify more data gathering
- Technical problems are the least of your concerns!
- Mistakes will happen, try to learn from them

So I've opened our proverbial kimono here and explained some of the mistakes we made, both big and small. So far our mistakes haven't added up to failure but we have spent a lot of time and money correcting them. When building your own software, you need to stay vigilant of the decisions you are making and the possible consequences of those decisions. Remember that technical design and code are usually far less important than nailing down your target market and ensuring your product is useable and adds more value than its cost. Finally, when you do make a mistake, ask yourself why did I make that mistake and try to generalize your answer so that you know how to avoid similar mistakes in the future.

Really dumb mistakes?

- Not on the record!
- mperham@gmail.com
- http://mikeperham.com
- http://fiveruns.com

Unfortunately for you I won't tell any dumb mistakes on camera. You can find me later and ask me or better yet buy me a beer and maybe I'll spill my guts. Thanks for your time!