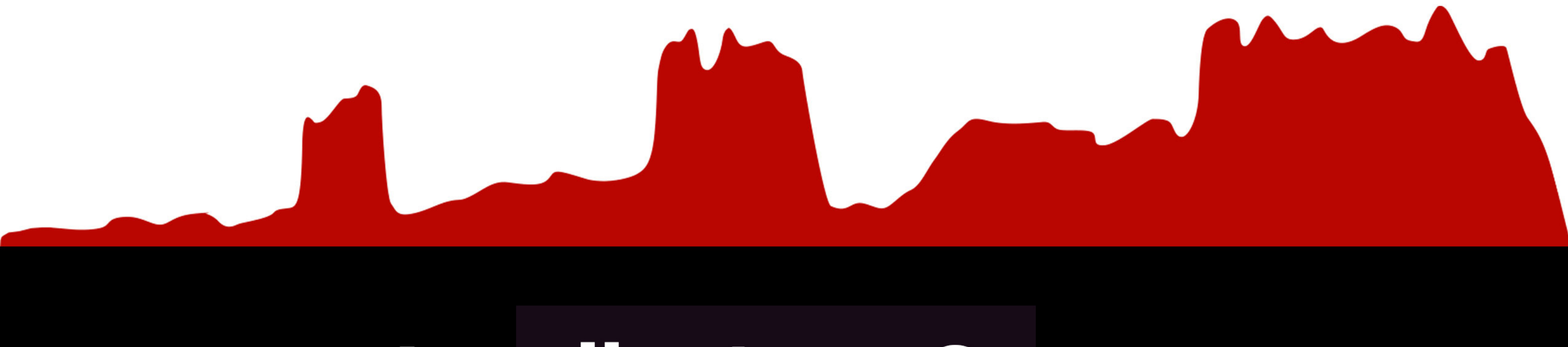
IDEAFLOW

How to Measure the **PAIN** in Software Development



Janelle Arty Starr

Idea Flow

How to Measure the PAIN in Software Development

Janelle Arty Starr

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

This version was published on 2019-05-14

Tweet This Book!

Please help Janelle Arty Starr by spreading the word about this book on Twitter!

The suggested hashtag for this book is #ideaflow.

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

#ideaflow

Contents

1. Losing Faith in Best Practices

Back in 2005, I was quite the hot shot. I was a pro at TDD. I was a master of refactoring. At least that's what I thought. When it felt like I understood, I just assumed I was right. There was never a doubt in my mind to challenge my own assumptions. I was as naive as I was confident.

I was working on one of the most disciplined software teams of my career. We had strict code reviews and acceptance processes for requirements, design, and testing. We had continuous integration, unit testing, and automation galore. We did all the "right" things.

Then I watched my project slam into a brick wall. We brought down production three times in a row, then didn't ship again for another year. Like the ground shattering beneath my feet, I lost my faith in best practices. Everything I believed about the right things to do was turned upside down.

Why did my team fail? What did I do wrong?

I could blame the risky choices I made, or the developers on the team who wrote crappy code. Regardless of why it happened or how completely terrible I felt, I'm absolutely grateful for the experience. It led me to change the way I look at everything in software development. It was an experience that changed my life.

1.1 The Crisis

My team was building a statistical process control system for semiconductor factories, one of the most sophisticated control systems in the world. Our software was responsible for reading all the measurement data from the tools and detecting processing errors. If something went wrong, the software could shutdown the tool responsible or stop the material from further processing.

I had been on the project for about six months, and we had just finished a development cycle. We had a lot of great new features, and the team was excited to get the release out. After passing all of our regression tests, we tied a bow on the release and shipped to production.

Later that night, we were on a conference call with IT while they installed the release. I could hear a guy screaming profanities in the background. Apparently, we had shut down *every* tool in the factory. Everyone was in a panic.

Fortunately, we were able to roll back to the prior release and get things running again, but we still had to figure out what happened. There was a configuration change that didn't quite make it to production. We fixed the problem, and assured our customers that everything would be okay this time.

Once again, we were back on the call with the installation techs, holding our breath as they started up the software. Ten minutes later, the *same thing* happened. As you can imagine, our customers were really upset. We didn't know what to say... Oops?

We went back to our tests, but couldn't reproduce the problem. We spent weeks trying to figure it out, to no avail. Weeks turned into months, and with about 10 people on the team sitting pretty much idle, management decided to move ahead with the next release. Everyone kept working like nothing was wrong, but we couldn't ship anything.

The reality was really sinking in. What if we couldn't figure it out? What would we do? Would we all be fired? I started to get mad. I didn't point fingers or yell at anyone, but on the inside, I told myself I wasn't the one who introduced the defect. Someone else screwed things up for our team. I didn't know what to do.

Third Time's the Charm

Eventually, we discovered the problem. There was a synchronized call in some of our multi-threaded code that was causing the system to crash.

By the time we figured it out, we had another release ready to go. We tested every crazy thing we could think of, scared to death of repeating the catastrophe. Eventually we had to cross our fingers and try again.

This time, there was a whole room full of people on the conference call. The installation tech pulled up a screen showing numerous charts that graphed the production activity. There was an initial spike during startup, but then everything looked okay. I went home that night relieved that things would finally be back to normal again.

It was about 3 AM when my phone rang. It was my team lead calling. He asked me about some code that I wrote and I knew exactly what happened.

I'd been working on performance improvements. I came up with this clever idea which flipped the architecture inside out to completely eliminate the most costly operations in our system. The team had struggled with performance for years, and my initial prototypes showed an order of magnitude performance improvement. My team lead, our customers, everyone was really excited.

If anyone could successfully rearrange the architecture after six months on the job, it was me. I did TDD! I tested everything! Unfortunately, my "improvements" introduced a memory leak and ground the system to a screeching halt. Our software crashed *again*.

To make matters worse, this time, the rollback failed. Not only had I brought down production, we couldn't get our broken software *out* of production.

I pulled out my laptop and started investigating the problem. A feature toggle! I told my team lead to update the configuration, but then the log file started filling with NullPointerExceptions! Apparently, I forgot to test the feature toggle. Could the situation get any worse?

I put together a patch fix to repair the broken feature toggle and we finally got our software running again.

There was nobody I could blame but myself. I felt sick.

Facing Failure

The next day, my boss called me into his office. He asked me what happened. I didn't want to cry at work, but that only lasted so long. I broke down sobbing. My boss quietly smiled at me, then gave me some of the best advice of my life.

"I know it sucks, but it's what you do now that matters. You can put it behind you, and try to let it go, or face the failure with courage, and learn everything that it has to teach you."

I wanted to crawl into a hole and hide, but I didn't. I took a deep breath. I wiped away my tears. That day, I learned that courage was a choice.

The next morning, everyone in the office was quiet. The team seemed to have the wind knocked out of them. My team lead brought in breakfast tacos to ease our stress. One of my teammates quietly poured everyone a cup of tea, and we all just sat there silently for a while.

"Let's try and learn everything we can," I said, breaking the silence. "We have an amazing team… I know we can figure this out."

I had never seen a team come together quite like ours did after this tragedy. It didn't matter anymore whether our process was the "right way" to do things. We had to learn how best to fix things for our project and our team. There was no way to rewind time and make different choices.

We had to accept where we were in that moment – failure.

Turning the Project Around

My project went completely off the rails. We could no longer ship to production. We had tests that weren't catching our bugs, and code that was too difficult to change. Yet, we had tons of automation, disciplined best practices, and an incredibly smart and talented team.

We could have thrown up our hands and run away from the challenge. We could have blamed our predecessors for all the crappy code that they left us, but that's not what we did at all.

On the day of the crisis, we lost our faith in best practices. We had no idea what to do. But rather than be paralyzed by the experience, we gathered data and analyzed the problems, and learned our way to success. Everytime we failed, we stood back up, dusted ourselves off, and tried again. We fought our way to success.

After two long years and a lot of hard work, we were shipping predictable quality releases. We restored our ability to understand the software and rebuilt the trust with our customers. We did it!

In the end, it was a victory, but the victory was bittersweet. It took us hitting a tipping point before anything changed. We lost predictability and our releases were breaking down before we did anything about the problems. We waited until we had no choice but to pay attention.

It took us over two years and millions of dollars in expenses to turn the project around. We had a release that took over a year to get out the door. During that time, our customers were struggling with our software and needed features that we were unable to deliver. The delivery process completely broke down.

I could celebrate how our team escaped the tragedy, but the real lessons are in how we got there to begin with. *why we didn't solve the problems* earlier? In hindsight, we had certainly sensed the problems, but why didn't our solutions work? Why did we have to get to a tragedy before we were willing to question our strategy?

I could celebrate how my team of frogs escaped the boiling water, but the real lesson was in *why we did nothing* as the water heated up in the first place. What were we doing wrong? How did we eventually succeed? How could we have avoided this whole tragedy?

1.2 The Rewrite Cycle

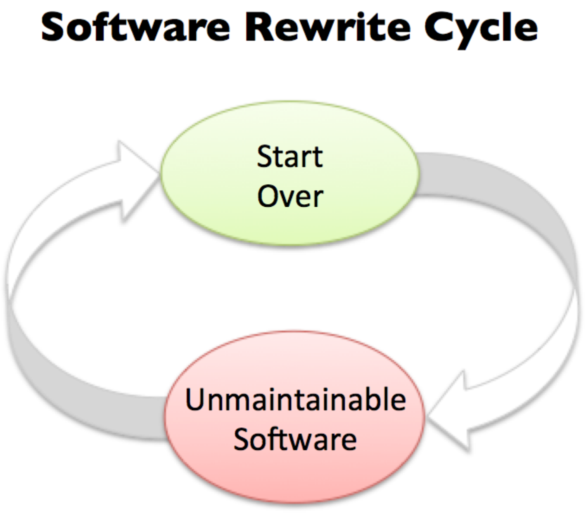
When I got into the consulting world, it was scary. Everywhere I looked, I saw teams repeating all the same mistakes that my team made. It was like looking through a window into the past.

We start off with the best intentions. We're going to write high quality code, that's low in technical debt, with lots of test automation, and short releases. We make a commitment to do things "the right way" this time.

Fast-forward to several years later, and we're sitting around a conference table discussing what went wrong. How did we accumulate so much technical debt? Should we rewrite the component, scrap the entire system and start over, or just deal with the problems and try to keep going?

In hindsight, we should have seen it coming. It's not like there weren't any clues. The outcome was clearly predictable, given our choices. We just weren't paying attention.

Then the rewrite cycle starts again… a new commitment, a new project, but this time we'll do it right.



Hindsight is a funny thing. We can look back at what happened and see where our decisions went wrong. If we did the project again, we'd know what to do differently. If we could only start over with a shiny new code base, we know we could get it right.

Why don't we get the results we're looking for? The challenges of software haven't fundamentally changed. We have certainly gotten better at it over the long term, but we've been talking about the same problems for over 40 years!

A Goal Without a Strategy

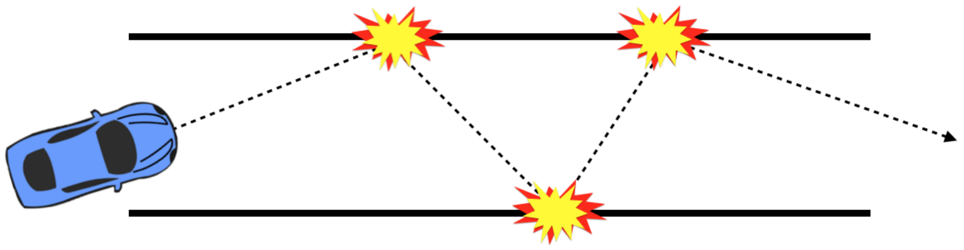
I was reading the book, Good Strategy Bad Strategy: The Difference and Why it Matters[1], and by the first chapter, I already had my answer:

> "A description of the goal is not a strategy." – Richard P. Rumelt

We talk about the importance of maintainable code and why it matters, but we don't talk about how the hell we're supposed to pull this off in the context of a business system. Our software projects don't operate on an island where maintainability is the number one priority. We're under constant pressure to deliver features as fast as we can, and maintainability takes a back seat.

Developers end up begging management for time to fix all the problems, but management doesn't understand the pain. When deadlines are slipping, and the project is behind schedule, management can't seem to take their foot off the accelerator pedal.

From the outside, it looks like we're trying to drive a car without a steering wheel. We line up the car's trajectory, based on our ideals, then close our eyes and floor the gas pedal. When we inevitably hit the wall and wreck the car, we have to build a whole new car to keep on driving.



We get stuck in the software rewrite cycle, because we don't *have* a strategy. We keep following the same dysfunctional organizational behavior patterns and keep repeating the same mistakes.

The Biggest Obstacles

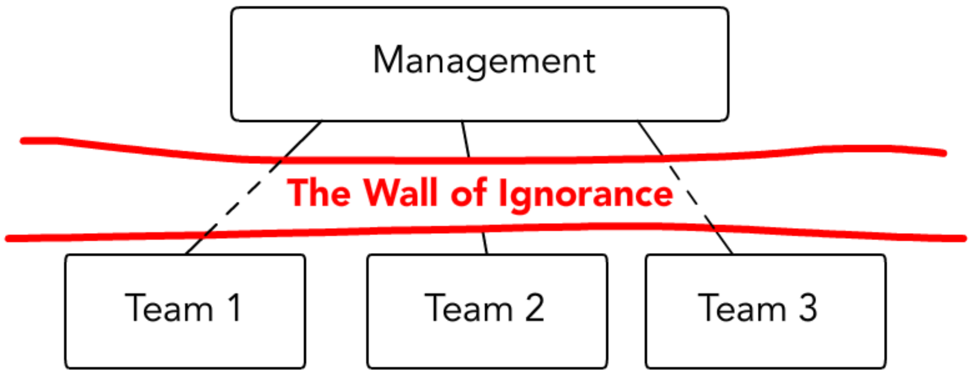
In Rumelt's words, "a good strategy is a specific and coherent response to – and approach for overcoming – the obstacles to progress."

[1]http://www.amazon.com/Good-Strategy-Bad-Difference-Matters/dp/0307886239

A good strategy is about leverage. It's about identifying the critical few things that will make the biggest difference, then focusing our efforts. So what are the primary obstacles that *cause* the software rewrite cycle?

> **Invisibility** - Feature work is highly visible. We can see new features in the product, interact with the features, and sell them to customers. The benefits are direct and measurable. The problems of confusion, misunderstandings, and mistakes are indirect and hard to quantify. We make our choices based on the benefits we see.
>
> **Lack of Control** - Developers try to explain the problems to the product owner, but the product owner doesn't understand. Developers try to explain the problems to management, but managers are under pressure to meet the deadlines. Even if developers know exactly what to do, change is not the developer's decision.

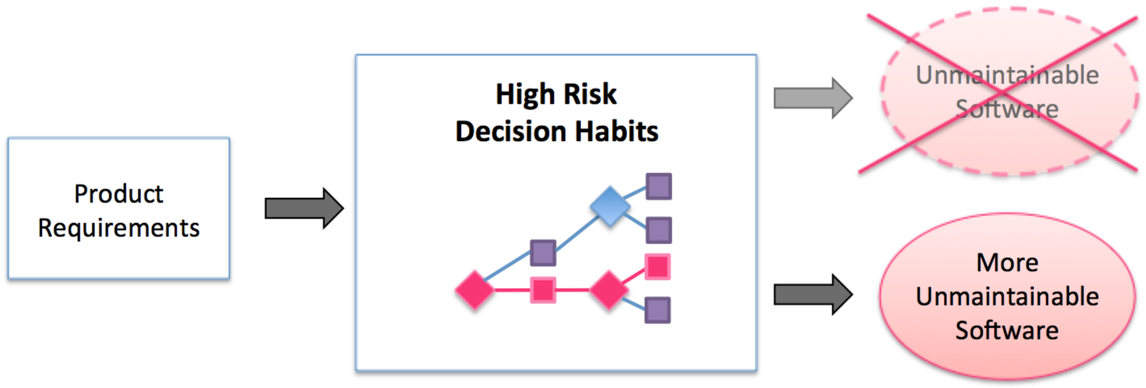
At a high level, our organization looks like this. Managers and developers might as well be speaking different languages.



Managers struggle with the lack of visibility. They can't see where they're driving the car. Developers struggle with the lack of control. They see the problems and yell, "We're going to crash!", but at the end of the day, the developers don't have control over the steering wheel. The broken feedback loops that lead our projects to failure are baked into the organization's design.

Our software is a reflection of our organization's decision habits. When we get in the habit of making high-risk decisions, we get unmaintainable software as a result. If we rewrite the software, but don't fundamentally change the way we make decisions, our problems just keep coming back.

We've been blaming technical debt for causing our problems, but technical debt is really the *symptom* of the problem.


Product Requirements
High Risk Decision Habits
~~Unmaintainable Software~~
More Unmaintainable Software

Any solution that's going to address the long-term challenges of technical risk management needs to address the obstacles that drive our projects into the ground. We need objective data that measures the impact of escalating risk. We need feedback that we can reasonably act on in the moment.

The Beginning of a Feedback Loop

When my project failed, we didn't have managers breathing down our necks telling us to ship features as fast as possible. At that point, the only thing that mattered was getting quality under control. We had a different challenge.

My team was trying to follow best practices, but we failed anyway. Starting over wasn't an option. The *only* way we could escape the tragedy was to learn. We found ways to measure the pain, understand the causes, and design solutions that worked. We took a project that had gone completely off the rails, and managed to restore predictability without starting over.

The key to turning the project around was visibility. Once we let go of our assumptions about best practices, that's when we started to learn. We didn't know the answers, so we had to look for them. We collected data, analyzed our mistakes, and identified the factors that increased quality risk.

We made the pain visible, then learned our way to success. It was the beginning of a data-driven feedback loop.

1.3 The Origin of a Learning Framework

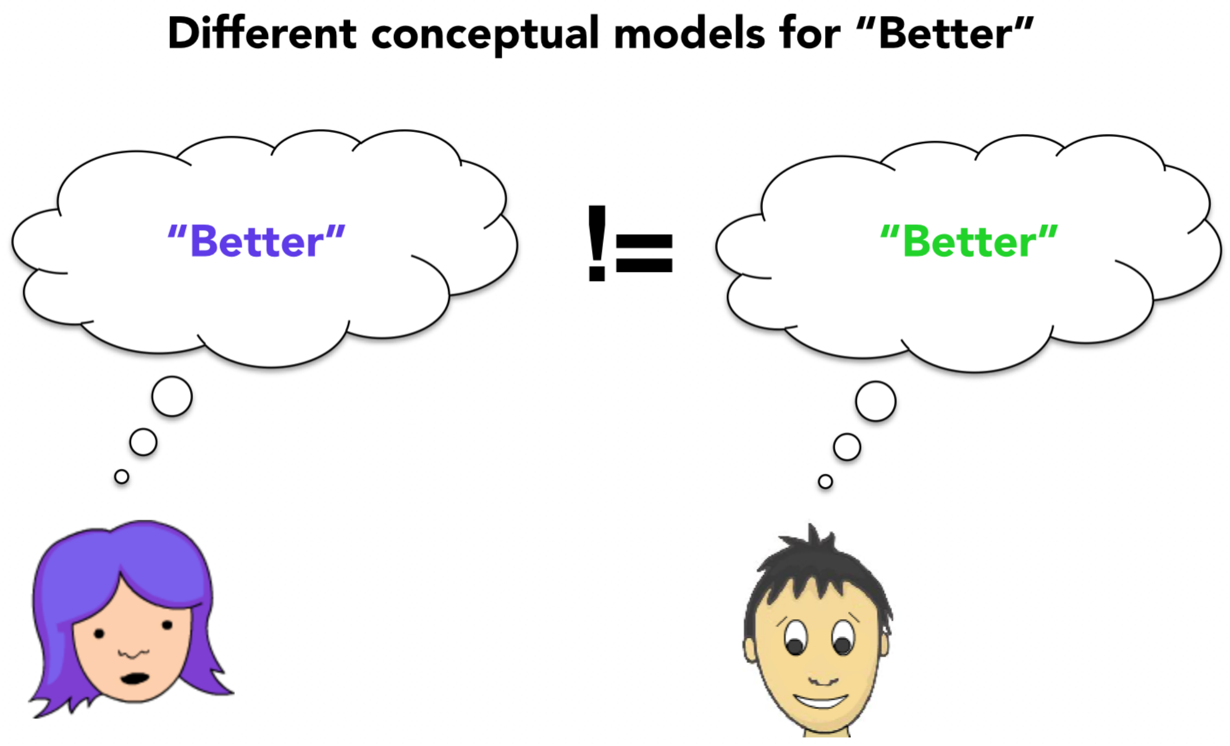
Several years later, when I was trying to share what I'd learned, I realized the bigger problem. No matter how I tried to explain the strategies of technical risk management, I couldn't seem to communicate my ideas.

I was trying to teach a developer on my team what I'd learned about good design. I walked him through examples. I explained the key concepts. He seemed to understand… then we got to the code review.

It was one of the strangest designs I'd ever seen. The object model made no sense to me and was really confusing and hard to understand, yet his variables and methods had descriptive names. He was trying to do what I'd said, but misunderstood the core idea.

I asked him about his thought process and what led him to the design decisions. He explained that he thought the code was more readable because he could see all the logic in one place. In his mind, he made the code "better" using the new practices I'd taught him.

As I listened to his explanation, I realized something important. It didn't matter how I explained the practices of good design, because he had a different conceptual model for the meaning of "better" that was based on a different set of experiences. He was optimizing for completely different things.



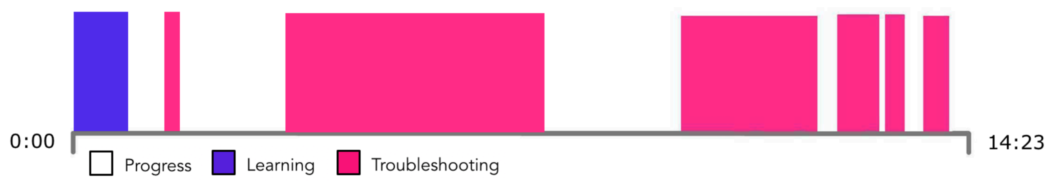
What did "better" really mean? How could I describe what I was aiming for? How could I put "better" into words?

On my process control project, we didn't change our beliefs until we had evidence that said our beliefs were wrong. We measured the pain, then learned how to avoid the pain. A better experience was a less painful experience. I was optimizing for developer experience.

I went home that night and looked at his code. It seemed like an obvious disaster. I needed a way to show him the problems with the design that were based on objective criteria. I needed to communicate my definition of "better".

Visualizing a Developer Experience

I started playing around with some likely enhancements that would build on top of his code. I recorded everything I did, how I struggled, where I got confused, and what mistakes I made. When I found myself confused, I tried to find the underlying reasons why. I visualized the friction in my experience on a timeline:



I imagined my interaction with the code as a stream of ideas, flowing between me and the software. If I had an idea in my head, and I was trying to put my idea into the software, there was friction that got in the way. If I was trying to understand an existing idea in the software, there was friction that made the ideas hard to understand. I visualized my Idea Flow.

I walked through the scenarios with the developer on my team, and showed him the data. I showed him the changes I had made, and the friction in my experience. I explained the causes of my mistakes. Then I gave him an alternative to his own design that resolved a lot of the pain. The new design made it easier to modify the code. The new enhancements flowed more easily into the software. He agreed that the new design was better and didn't have the same drawbacks.

The story of my experience with the code was the objective feedback that he needed.

A few days later, he was working on a new design, and called me over to ask a question. He was working on writing a unit test and asked me if I thought the test was failing in a "good way." It was funny how he said it, but I knew what he meant. He was thinking about the experience of the next developer that had to work with his code. If his test failed, would the test show the developer what was wrong? I asked him questions about various hypothetical scenarios. As we talked, he made improvements to the tests.

I knew he was starting to see. He was thinking about how the software would change over time, and how his decisions might cause developer friction. By recording my experience with the code and sharing my struggles, he was thinking about how to make future Idea Flow "better".

That was the beginning of Idea Flow Learning Framework. I started teaching developers how to visualize their coding experience and get objective feedback on the consequences of their decisions. As I learned about problems and techniques to optimize developer experience, I codified the lessons into patterns and principles.

I started using my Idea Flow Mapping technique as a way to teach developers how to learn.

Breaking the Rewrite Cycle

Once I solved the visibility problem, I realized the bigger potential. I could use Idea Flow Mapping to measure the pain on a software project, and help the team explain the problems to management. I could teach managers about the nature of software risk, and how to avoid driving their projects into the ground.

I started aggregating Idea Flow data and using the learning framework as a strategy for long-term technical risk management. By measuring the pain, understanding the causes, and focusing on the highest leverage improvements, it was a strategy to turn around failing projects.By responding to the early indicators of problems, it was a strategy to keep software projects from spinning out of control.

Visibility created a means for project-level control. Managers have been driving their software projects blind-folded. The only feedback a manager has for making decisions is the developers complaining about their invisible pain. With looming deadlines and no understanding of the risks, our managers don't know what to do.

We crash the car over and over again, because we don't have an effective feedback loop. In order to steer our projects, we need visibility and control. We have to *see* the problem, then *steer* around it.

We can fix the organizational problems by repairing the broken feedback loop.

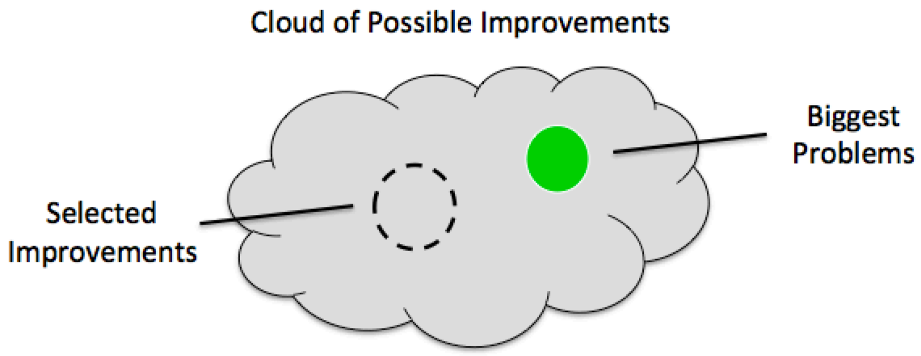
The Recipe Isn't Enough

When we teach best practices, we teach developers what to do, but not how to think for themselves. We don't teach developers how to see the pain, understand the problems, and make decisions on their own, we teach developers how to follow the recipe.

When our strategy fails, we don't know what to do. If we make our project more best-practice-like, we assume things will get better, but it doesn't work. When we don't understand our problems we tend to spend tons of time making improvements that don't actually make much difference.

One of the hardest lessons for me was realizing that all the tools and best practices that I depended on, what I believed were the "right" things to do, were more often a distraction than a path to success. In order to reach success, I had to stop believing that I knew what to do. My team failed because we didn't understand the problems.

When we were following best practices, we never felt like we were doing anything wrong. Yet, by focusing on best practices, we filled our improvement list with all the wrong things. We spent tons of time working on improvements that didn't really make much difference. We completely missed our biggest problems because we were too busy making our project more best-practice-like.



We had to bring down production three times in a row before we started to question our strategy!

The Questions Stay the Same

I believe the root cause of project failure in our industry is our inability to teach the lessons of experience. We try to pass on our knowledge through best practices, but the knowledge is lost in translation. We don't arm people with the skills they need to adapt in the face of complex challenges in a world that's changing faster than ever.

In the words of Keith Cunningham, "The solutions keep changing, but the questions stay the same."

> How do we sustain understanding as complexity grows and the team changes?
>
> How do we mitigate the risk of constant change?
>
> How do we design around our human limitations?

I don't teach best practices anymore. I teach developers how to think and make decisions. I teach developers how to recognize the pain, evaluate their situation, then systematically optimize Idea Flow. I teach developers, teams, and organizations how to learn.

I don't have a magic solution, but I do have a new approach. The ideas presented in this book are a strategy for learning what works. It's a way to identify the causes of pain, dive head-first into the challenges, then systematically learn our way to success.

This book is about my journey in learning with a data-driven feedback loop.

I Visibility

2. Technical Debt is Not the Problem

After my team brought down production for the third time in a row, it was pretty clear that our strategy for managing quality was ineffective. We spent tons of time testing the code, refactoring the code, and doing our best to follow best practices, but at the end of the day, none of it seemed to matter.

It was the day after the production catastrophe at our morning standup meeting. Our project manager was waiting impatiently for us to finish our daily updates. “I don’t care what you guys do to solve the problem, but you need to FIX THIS. I’ve been apologizing to people all morning. This can NEVER happen again.” He promptly left the room and slammed the door.

“So what are we going to do?” My teammate asked. “Clearly the tests aren’t catching our bugs. More automation?”

We tested every scenario we could think of, but the bugs were always in scenarios that we *didn’t* think of. It was annoying. It’s not like we weren’t careful. The software was extremely complex.

“I don’t think the problem is the tests, it’s the application code,” I replied. “We should have been more focused on reducing technical debt. I’ve been saying it all along.”

It seemed like technical debt was building up in the system and causing us to make mistakes. We were chipping away at the debt, but the pressure to deliver features never let up. There was never enough time to fix all the problems.

The project was about 10 years old and there were parts of the system that nobody understood. There were several things we wanted to rewrite. We had a few 5,000-line classes that made everyone cringe, and a long list of technical debt to fix.

My teammate was frustrated by my response, “That’s all well and good, but we can’t change the past! What are we going to do *right now*? How are we going to get out of this mess we’re in?”

It was a valid question. We couldn’t start over and make different decisions. The best we could hope for was a better assessment of the risks. Hmm. *Risk* assessment. That gave me an idea.

“If the technical debt is causing us to make mistakes, then if we make changes in areas of code with *more* technical debt, we’d be more likely to make mistakes, right? We could build a tool to detect high-risk changes that let us know where extra testing was needed.”

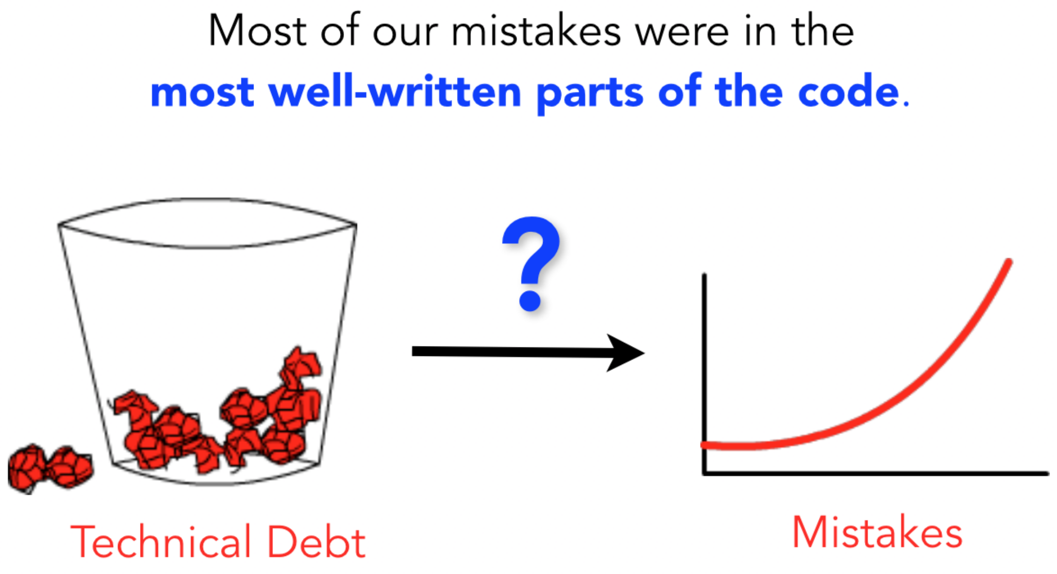
Everyone was excited about my idea.

2.1 More Questions Than Answers

I cobbled together a static analysis tool that could detect our high-risk changes and provide a risk assessment for each release. I had several years of code history and defects for past releases, and I figured I could use the data to show how technical debt was causing our problems.

I expected to see mistakes more often in high-risk code, but that's not what I found at all.

Most of our mistakes *weren't related* to the health of the code. I couldn't find any significant correlation with mistakes, no matter what I did with the data. A lot of mistakes were in the most well-written parts of the system. I tried everything I could to show the impact of technical debt, but the data just didn't add up.


Most of our mistakes were in the
most well-written parts of the code.
?
Technical Debt
Mistakes

At the time, I didn't know how to make sense of the data. I started going through each defect, one at a time, and talking with the developers to find out what had happened. A lot of the causes were similar. An edge case was missed. Something was forgotten or misunderstood. It was hard to think through all the required changes. These problems sounded like they could be caused by technical debt, but there was clearly more to it than that.

I found one correlation in the data. We made significantly more mistakes when we modified code that was written by others. We were working on a large application that had a lot of different hands in it over time. Were our mistakes just caused by a lack of familiarity?

I wanted a simple answer, but there wasn't one to be found. Somewhere in the process of interacting with the code, something was going wrong. The problem wasn't in the code. The problem wasn't the technical debt. All I knew was that I didn't understand the problem.

What Causes Us to Make Mistakes?

Reducing technical debt made a lot of sense. Some of the code was poorly designed and needed improvement. Whenever we had to work with the code, it was painful. We could reduce this pain by reducing the technical debt in the code.

Now I had data that didn't support such reasoning. I needed an alternative explanation.

I started writing down everything that was going through my head as I worked. Every time I made a mistake, even a small one, I wrote it down. I kept track of how long it took to troubleshoot each problem, then looked for patterns in what made troubleshooting difficult.

A lot of my mistakes were simple things. Sometimes I didn't read the code very carefully and made bad assumptions. Other times I transposed arguments, forgot to do things, or just didn't do what I intended. I started to see patterns in the ways I made mistakes in the context of my *interaction* with the code. For example:

- **Familiarity Mistakes** - Working with code that I don't know is always more mistake-prone, even when the code is not complex.
- **Stale Memory Mistakes** - Sometimes, I make wrong assumptions about how the code works, because the code doesn't work the way I remember.
- **Semantic Mistakes** - Other developers have ideas and meanings for words that differ from my own. Sometimes I misinterpret them.
- **Scanning Mistakes** - When I try to figure out what matters and what doesn't, I quickly scan through a lot of different code. Sometimes, I miss critical details.
- **Copy-Edit Mistakes** - Sometimes, when I copy a block of similar code and then edit the parts to be different, I miss one of the changes.
- **Transposition Mistakes** - I frequently flip-flop parameters and if-else conditions when I code.
- **Failed Refactor Mistakes** - When I'm trying to refactor the code, sometimes I accidently change the behavior.
- **Tedious Change Mistakes** - Sometimes the changes are time-consuming and tedious, and I just miss things because I'm half asleep.

What I discovered was simple, but remarkable. Most of my mistakes had nothing to do with technical debt. My mistakes were caused by human factors in the work.

Getting Stuck on an Idea

No matter how much evidence I found that technical debt wasn't causing our problems, it was hard to let go of the idea. I kept trying to find something in the code to blame. It was as if I had already decided on the conclusion and was searching for evidence to support it. This was confirmation bias[1] at work.

Technical debt constrained my thinking. I couldn't see how *my actions* affected the situation. I couldn't see myself as part of the system.

Several years ago, I read a book called Metaphors We Live By[2]. I was amazed by the power of metaphors in our thinking. Our ideas are shaped by the metaphors in our culture. We don't even realize how many of them affect us.

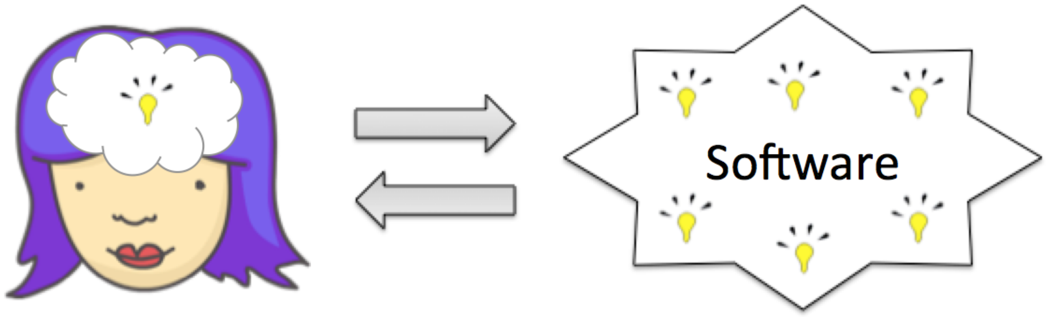
[1]http://en.wikipedia.org/wiki/Confirmation_bias
[2]http://www.amazon.com/Metaphors-We-Live-George-Lakoff-ebook/dp/B009KA3Y6I

Technical debt was one of those metaphors. It affected how I saw the causes of our problems, and made it hard for me to see anything else. I needed a way to focus on the human factors in the work, and stop looking for problems in the code. I needed a new metaphor.

What is Idea Flow?

The pain I experienced wasn't an attribute of the code. The pain occurred during the *process* of understanding and extending the code. Through an interactive process of learning, experimenting, and working out the kinks, I had to struggle through the pain to find to a solution.

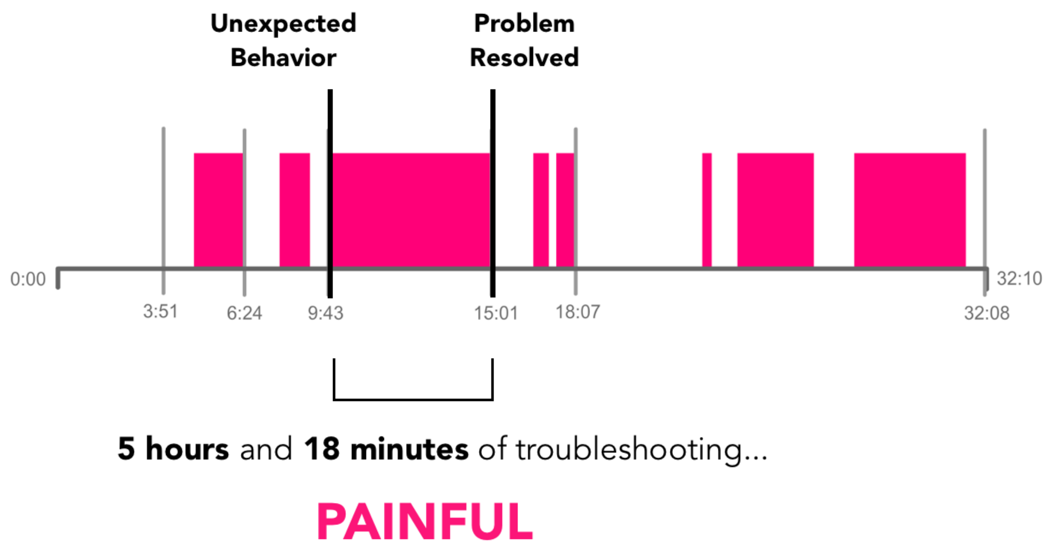
Idea Flow is a metaphor for the human interaction in software development. Think of Idea Flow as the flow of ideas between the developer and the software.



Suppose I have an idea in my head about how parts of the software should work. I can move that idea into the software, molding the code as if it were clay to represent the idea in my mind. If another developer were to read my code, the shape and character of that same idea would come to life in their mind.

Software is a medium for ideas. The ease with which I can get an idea that's represented in the software into my head is understandability. The ease with which I can add a new idea to the software is extensibility. **Friction** is everything that gets in the way of the flow of ideas. Friction is what causes our mistakes.

By visualizing my interaction with the code, I could see my struggles with understanding. All the time I spent troubleshooting, I mapped in red. This is an Idea Flow Map:



The code didn't need to be full of technical debt for me to struggle with mistakes. I built a mental model for how the code worked. I made assumptions and set expectations that turned out to be wrong. I made mistakes because I was human.

If I wanted to reduce the friction I experienced, I needed to understand the human factors of software development.

Metaphors are Filters

Metaphors are simple, but have a powerful effect. They affect the details we see. They change the way we think. Metaphors are the rose-colored glasses we see the world through.

The more familiar we are with something, the harder it is to look at it differently. I'll never forget the day my dad showed me something new and beautiful, within something I'd seen every day.

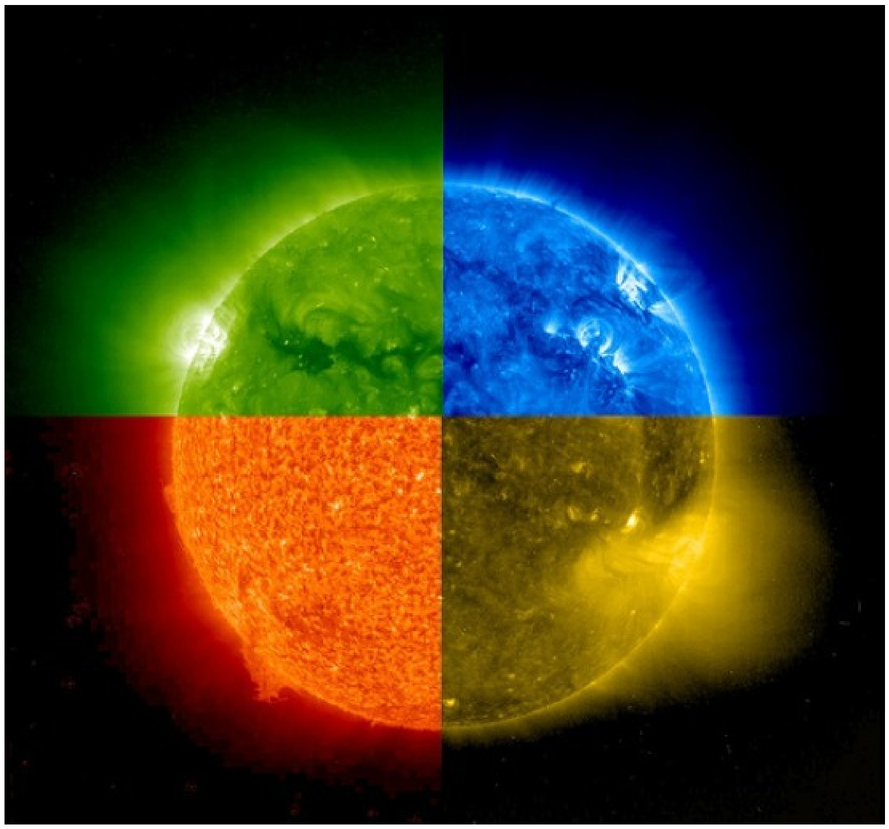
My dad loves the stars. He lives in central Oregon, which is one of the best places on the planet for stargazing. On a clear night, you can see thousands of stars in the Milky Way, streaked across the sky. It's amazing. One night while I was visiting, he set up his telescope on the deck to give me a tour of some noteworthy star clusters and galaxies. I'd read about stars and seen photographs, but I was mesmerized. Seeing a galaxy with my own eyes, I felt like a tiny speck of dust. The universe was beyond anything I could ever understand or imagine.

My dad insisted we look at one more star, but I was so exhausted that I couldn't keep my eyes open, so I stumbled off to bed. I hadn't gotten but a few hours of sleep when my dad excitedly knocked at my bedroom door. More stars? I could see the sunlight already peeking through the curtains. What was he thinking?

I followed my dad outside, squinting in the brightness. Then I saw his telescope pointing directly at the sun. One more star. Staring at the sun through a telescope seemed like a really bad idea, and I'd already seen the sun nearly every day of my life. He rolled his eyes at my hesitation, and urged me to take a look.

With the telescope's filter, the sun was a radiant orange on a black background. It had a skin like an orange peel with a twisting current at its center, and a few scattered bright spots across the surface. Ok, that was cool. No matter how many times I'd seen the sun, I'd never seen it like this. There were so many details that I'd never have seen without the filter.

My dad had me step back while he replaced some parts on the telescope, and had me look again. This time most of the sun was black, but with big swirls and arcs exploding from the sun's surface. The details were amazing. The filter hid the orange-peel texture, but highlighted a whole new set of features I didn't see before.



Filters have a powerful effect. Even with something familiar that we see every day, with a filter we can see it differently. By hiding details, filters highlight other details. They create a path to understanding by allowing us to focus on just one part at a time. Filters allow us to see a bigger picture.

Metaphors are filters that we use to understand our world, including software. We simplify what we experience by ignoring details that are hidden by our filters. Just like in a telescope, our filters highlight some aspects of software and hide others. Our filters create blind spots in our thinking.

To see something new, we have to let go of our current filters and be willing to try on a new one.

2.2 An Alternative to Technical Debt

Technical debt had me stuck on the idea that our problems were somewhere within the code. I thought I could find the causes of our mistakes by analyzing our software. I ignored all the signs that pointed somewhere else, and kept searching for a way to confirm my beliefs.

Don't get me wrong. Technical debt is a very useful metaphor. On many projects, there's constant pressure to produce features as quickly as possible. The people making the decisions usually have no idea of the consequences of the time pressure. Technical debt helps us communicate the long-term trade-offs of these decisions. It's important to keep the code in a healthy state.

However, if technical debt is supposed to represent the growing friction we experience, what are we missing with this metaphor?

No Time Pressure != No Debt

The obvious solution to avoiding technical debt is to avoid time pressure, and to pay off our debt as early as possible. If we take the time upfront to avoid shortcuts, we shouldn't have problems later, but we do. Often, our interest payments have nothing to do with taking shortcuts.

For something to be a shortcut, we have to realize we're doing something that is less than ideal, but we regularly make poor decisions that have significant costs without having any idea they are poor decisions. We don't realize how difficult our code is to work with until we hand it off to someone who isn't familiar with it. We don't realize we've created a painful development experience until developers start complaining about it.

Software development is fundamentally a discovery process. We will *always* make bad decisions. Most bad decisions we discover in hindsight. Avoiding technical debt is not as simple as avoiding shortcuts.

Variability in Cost

In financial debt, an interest payment is typically a small portion of the original debt, that can grow substantially over time. However, we wouldn't borrow $100 and pay $1,000,000 in interest, not even from a loan shark. If we borrowed the same amount twice, we would expect to pay a similar interest rate. However, technical debt doesn't work that way.

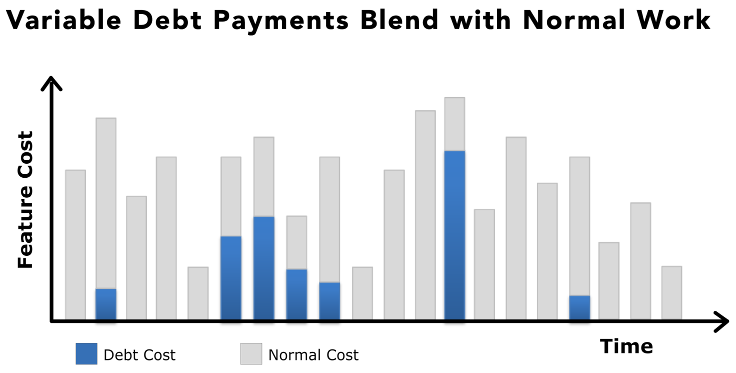
Suppose we borrowed the same amount of time for two different technical debts. In one case, we didn't pay any interest, because the impacted code never had to be touched again. In another case, we had to pay our time investment several times over, and the costs skyrocketed.

The reasoning we use to make decisions about financial debt can't be easily applied to software development. We usually have little idea in advance of how much the debt will cost us. Even in the middle of paying hefty costs, it's not obvious how much we're paying.

Variability in Time

With financial debt, interest payments are regular and consistent over time. With technical debt, that couldn't be further from the truth. Technical debt only has a cost impact when we have to understand or modify the affected code. If a new feature doesn't intersect with the technical debt, it doesn't impact us at all. It's not uncommon for code to sit idle for years with no one needing to read it.

The amount of the interest payment is also variable in time for the same debt. When we work in an area of code, the better we need to understand the code, the more time it takes. Sometimes technical debt only impacts us a little. Other times, we have to spend countless hours trying to understand the code. This variability makes it hard to see any problem from the outside. There is no uniform increase in feature costs that makes the problems easy to see. The interest payments blend in with the normal work.



Beauty is in the Eye of the Beholder

What we see as technical debt is subjective. Some code problems are hard to understand for almost anyone reading them, but a lot of things aren't that black and white. Most would agree that a 400-line method is hard to read, but what about using a visitor pattern versus a for loop? Or using inheritance versus a strategy pattern? Or a state machine versus conditional retries in try-catch blocks?

I once worked with a guy whose brain seemed to be wired differently from everyone else on the team. He was brilliant, but his up was everyone else's down. Code that looked elegant and clear to him often seemed confusing to others on the team, and code that was clear to others looked messy and confusing to him. He always wanted to rewrite the code written by others, and vice-versa. I think this same problem happens on most teams, but to differing degrees.

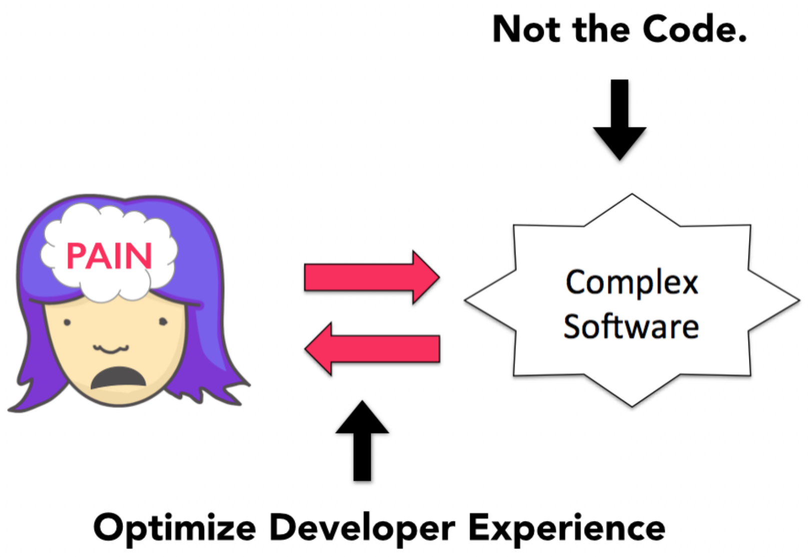
The code we write always looks clearer to us. It looks understandable because we understand it. There is usually some desire to make changes to code that other developers write to make it clearer to ourselves. The more different our wiring, the more foreign the code looks, and the stronger the desire to "fix" the code.

This isn't just a matter of ego. The same code can be clear to one person and confusing to another, resulting in accumulation of technical interest payments when the second person needs to work with the code. There's no absolute definition of good code that transcends the fact that multiple humans have to understand it.

Humans are Part of the Problem

If half the team is able to work with and extend the code easily, and others struggle with the code for hours, does the code contain technical debt? What if only one person is able to work with the code easily? Does that mean the code has technical debt? If we're looking for our interest payments somewhere within the code, we are missing a crucial aspect of the problem.

The problem we're trying to describe occurs during the *process* of understanding and extending the software. If there is no human interacting with the software, the problem goes away.



Technical debt suggests the problem is a thing, so we look around for a thing to blame. We see painful nouns, when we should be seeing painful verbs. If we want to reduce the pain that occurs when humans interact with code, we can't just ignore the humans.

2.3 Idea Flow is a Conversation

Let's look at the process of software development another way. Remember, Idea Flow is the flow of ideas between the developer and the software. Our interaction with the code is similar to a conversation.

Think about what happens when you are trying to share an idea with a friend. Your goal is to put the same idea that's in your head into your friend's head.

However, an idea isn't a tangible thing that we can touch, taste or see. We can't put an idea into a box and hand it to someone. When we communicate an idea, we can't be sure that what we sent was the same idea that was received. Our prior experiences, our perceptions, and the meaning we associate with words all affect the content of the idea we receive.

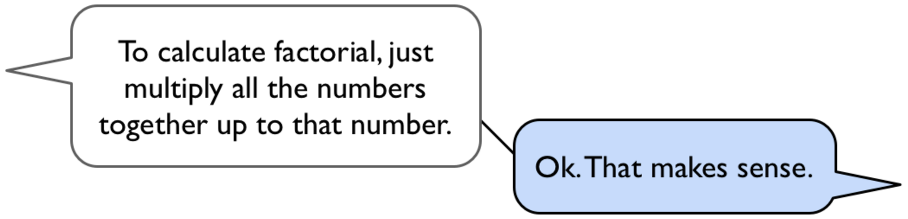
Even though I can't really tell if my friend understands an idea, I have ways of evaluating her understanding. I can ask questions, have her explain the idea back to me, or simply take her word that she understands. As long as her comments make sense to me, I assume she received the message I sent. If she makes a comment I find surprising, I can try to diagnose where the communication went wrong, and fix it.

With software, it's not really all that different. We start with an idea in mind that we want to communicate to the software. We tell the idea to our software through coding. To see if the software understood, we ask it a question by running a test and reviewing what it says.

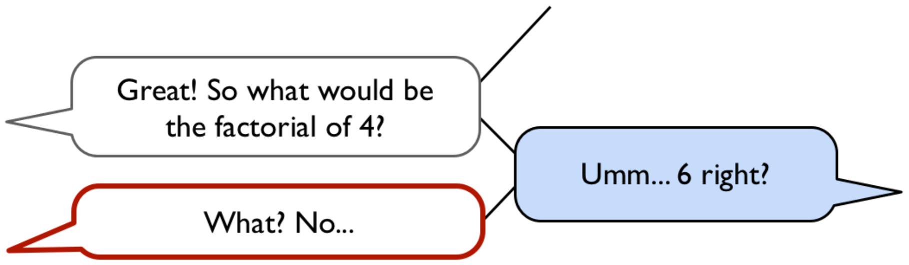
If the software doesn't say anything unexpected, we assume we communicated successfully. If the software says something that surprises us, we have to diagnose where the communication went wrong and fix it.

A Conversation with Code

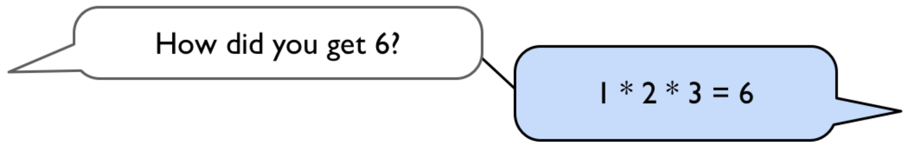
Let's walk through an example of how a conversation happens in software. Imagine that the code is a friend of ours who is learning a new concept.


To calculate factorial, just multiply all the numbers together up to that number.
Ok. That makes sense.

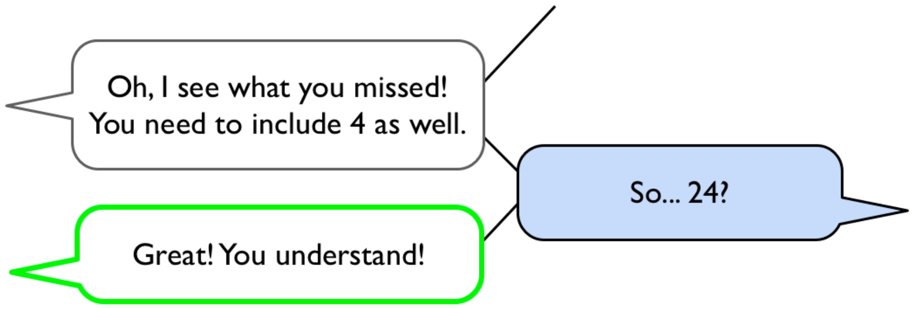
Did our friend understand the idea? It seems like it, but we aren't really sure. So we ask questions.



This answer was surprising. Clearly something went wrong, so now we have to diagnose the problem.



Now we know how our friend misunderstood, so we can correct the problem.



Finally we get the answer we expected, and we're confident that our friend understands.

By paying attention to the interactions during development, we can recognize these conversations with the code. What causes friction in software development is a *breakdown in communication.*

Confirmation and Conflict

In the example, I highlighted two of the responses. The green outline represents an observation that confirms our beliefs, and the red outline represents an observation that conflicts with our beliefs.

Whenever I read the code, run a test, or run the application, one of two things can happen. Either the observations agree with my existing understanding of the software, or they don't, confirmation or conflict. Confusion doesn't occur because our tests are red or green, but because our *expectations* are confirmed or violated.

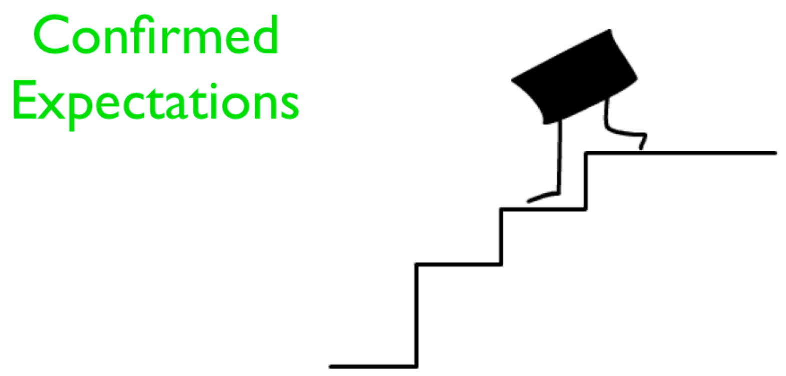
Whenever we see something unexpected, we stop working on the feature and try to figure out what's wrong. This disruptive conflict pattern is how we experience software friction. The more time we spend diagnosing these problems, the more friction we experience.

Prediction and Conflict

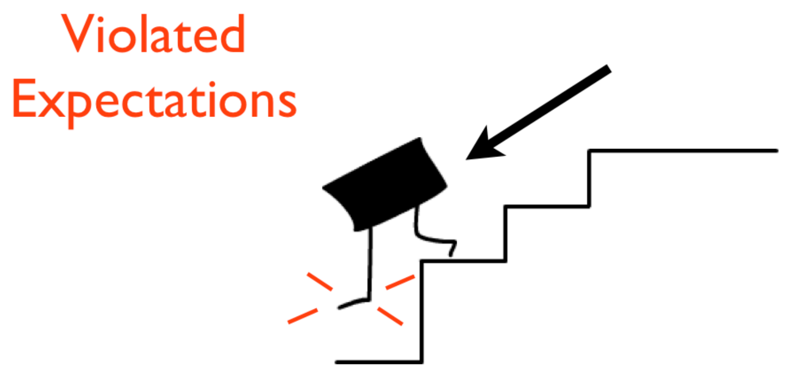
After I presented Idea Flow in the Lean community[3], a friend of mine urged me to read Jeff Hawkins' book, On Intelligence[4]. The experiences of confirmation and conflict, I discovered, are wired into our biology.

[3]http://www.meetup.com/Lean-Software-Austin/events/75941862
[4]http://www.amazon.com/Intelligence-Jeff-Hawkins/dp/0805078533

Imagine you're walking down a flight of stairs. Everything feels fine, and you're thinking about something else. Your expectations are confirmed.



Then suddenly, one of the steps isn't where it was supposed to be. Every part of your body is suddenly paying attention to that moment. Are you going to fall? Can you grab the railing? Do you need to brace yourself? When your expectations are violated, you experience a conflict.

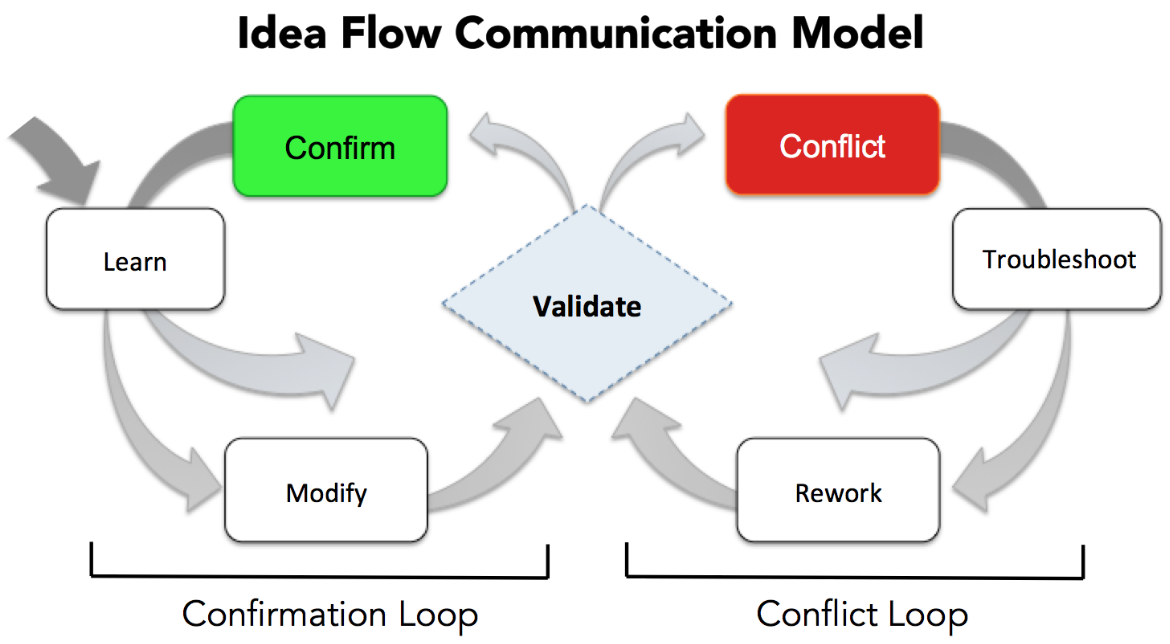


Our memory works like a big prediction engine. Our sense of the world is constantly compared against our existing understanding to see if anything seems out of place. When we sense something, our brain predicts what it expects to see next. If that prediction is violated, it tries again. If our brain can't find any predictions that make sense, we experience a conflict.

Our brains are hardwired to work this way. As long as everything we observe matches our predictions, our consciousness isn't bothered with the details. But as soon as something unexpected happens, we can't help but pay attention. This is the same thing that happens when we develop software. A conflict is just a *violated* prediction.

The Idea Flow Communication Model

Software development is a process of writing a little code, then working out the kinks, over and over again. It's a series of conversations with the code.



Every time we validate the code, we take one of two paths. A confirmation loop makes progress toward the completion of the feature. We read the code and learn how it works, then modify the code to do something new.

A conflict loop occurs whenever the code doesn't work as expected. We have to troubleshoot the problem and resolve our concern before we can return to making forward progress.

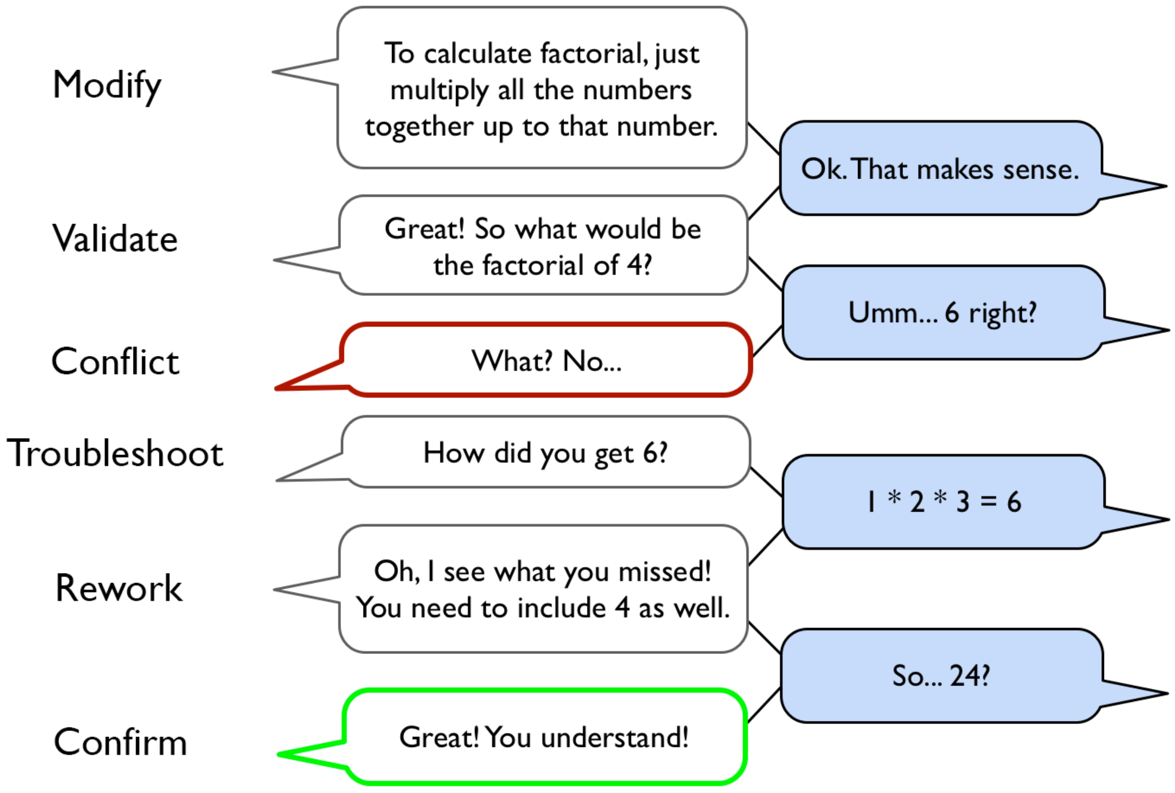
> **Learn** - With any task, we have a goal in mind. We have to read the code and build a conceptual model of how it works, in order to develop a strategy for reaching the goal.
>
> **Modify** - Next, we modify the software to do something new. This is where all of the coding happens to complete the feature.
>
> **Validate** - After we finish an increment of code, we run experiments to observe and validate the behavior.
>
> **Confirm** - If the behavior matches our expectations, our understanding of the software is confirmed. We feel as if our understanding is stable, like we "get it."

Conflict - If the behavior violates our expectations, we experience a sense of conflict. Our attention snaps immediately to reconciling our understanding, and we usually have trouble focusing on anything else until we resolve our concern.

Troubleshoot - Once in conflict, we have to search for information that will help us make sense of the unexpected behavior. Troubleshooting usually means running more experiments.

Rework - Once we identify the cause of the unexpected behavior, we usually have to rework the code so it matches our intentions. If there's a bug, we need to fix it.

Looking back at our example coding conversation, we can see how the loop pattern appears.


Modify
To calculate factorial, just multiply all the numbers together up to that number.
Ok. That makes sense.
Validate
Great! So what would be the factorial of 4?
Umm... 6 right?
Conflict
What? No...
Troubleshoot
How did you get 6?
I * 2 * 3 = 6
Rework
Oh, I see what you missed! You need to include 4 as well.
So... 24?
Confirm
Great! You understand!

We can't write new code on a foundation of confusion. Likewise, to extend the software to do something new, we have to build from a place of understanding. A conflict loop is also a learning loop. We can't learn if our understanding isn't challenged.

2.4 Optimizing Developer Experience

Over the last few years of studying Idea Flow, I've learned more about software development than ever in my career. Once I started measuring my experience with the code, I had a feedback loop for optimizing developer experience.

The circumstances of my mistakes were always different, but I started to see patterns. They were simple patterns at first, like tendencies I had when I wrote code. For example, I transposed if/else blocks *a lot*. I would copy and edit code instead of typing it from scratch when it looked similar to something nearby. I tended to space out and make mistakes when the work was tedious.

Eventually I started to recognize bigger patterns. I learned techniques to reduce troubleshooting time, minimize learning effort, and reduce the risk of mistakes. I worked on improving my testing technique, mistake-proofing my design, and optimizing for the ease of understanding.

Our struggles with troubleshooting aren't simply caused by chance. The pressures on the project, our trade-off decisions, and how we decide to approach the work all contribute to the friction we experience today.

Some of our decisions increase the risk of mistakes, even though we're trying to make things better. We remove duplication at the expense of increasing complexity. We break up the code trying to make it easier to read, but then we split up details that need to be understood together. We are often the ones making the work more mistake-prone.

Making good decisions is difficult. Software trade-offs are not black and white. Removing duplication isn't always a good thing. With objective feedback on the consequences of our decisions, we can find our way to better in a vast world of gray.

When my team brought down production three times in a row, it wasn't because we weren't following best practices. It wasn't about unit testing, technical debt, or shorter releases. We were trying to do all the right things and constantly working on improvements, but we weren't solving the right problems. We didn't see the patterns even though they were right in front of us. We were paying attention to the wrong details.

When we finally turned the project around, it wasn't because of our test framework or the new software design. We learned how to ask the right questions. We started discussing the challenges of understanding the code and the ways we misunderstood the behavior. We started working on solving the human problems. We made the system easier to understand. We built tools that made it easier to learn.

Software, at its core, is an expression of human thought. If we want to improve, we need to understand what we're doing wrong, which means understanding the human factors in the work. Once we start asking the right questions, we start finding the right solutions.