Read This First

This book has no introduction.

Yes, you read that right. Now, you might immediately ask yourself, “Why doesn’t Jeff’s book have an introduction? Did he forget to write it? Is he beginning to slip after all these years?! Did the dog eat it?”

No, I didn’t forget to write an introduction to this book. And, no, I’m not beginning to slip. At least I don’t think I am. And my dog didn’t eat it (although my daughter’s guinea pig looks suspicious). It’s just that I’ve long believed that authors spend too much time convincing me I should read their book, and a great deal of that convincing lives in the introduction. The meat of most books usually doesn’t start until Chapter 3. And I’m sure it’s not only me who does this, but I usually skip the introduction.

This book actually starts here.

And you’re not allowed to skip this because it really *is* the most important part. In fact, if you only get two points from this book, I’ll be happy. And those two points are right here in this chapter:

- The goal of using stories isn’t to write better stories.
- The goal of product development isn’t to make products.

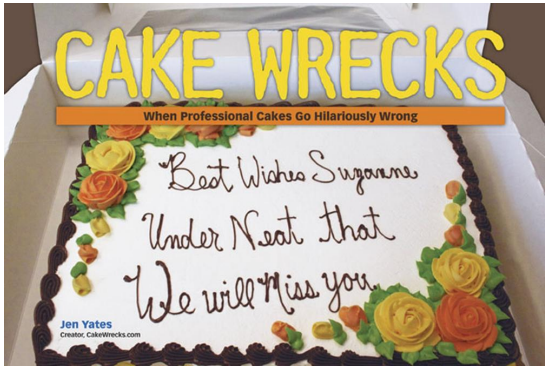
Let me explain.

The Telephone Game

I’m sure you remember when you were a kid and you played this weird “telephone game” where you whispered something to somebody, who whispered it to someone else, and so on around the group, until the last person reveals the totally garbled message and everyone laughs. Today, my family still plays this game at home with my kids around the dinner table. Note to parents: this is a good activity to occupy kids bored with adult dinner conversation.

In the grown-up world, we’ve continued this game—only we don’t whisper to each other. We write lengthy documents and create official-looking presentations that we hand off to someone, who proceeds to get something completely different out of it than we intended. And that person uses that document to create more documents to give to different people. However, unlike that game we played as kids, we don’t all laugh at the end.

When people read written instructions, they interpret them differently. If you find that a little hard to believe (it’s in writing, after all!), then let me show you a few examples of instructions gone very, very wrong.



This is the cover of Jen Yates's book *Cake Wrecks* (Andrews McMeel Publishing). (Thanks to Jen and John Yates for supplying these.) The book sprang from her wildly entertaining website, *cakewrecks.com*. Please don't go there if you don't have at least an hour to waste. The site shows photos of oddly decorated cakes that defy explanation—but Jen explains them in spite of that. Now, one of the recurring themes in both the site and the book is misinterpreted requirements. But of course she doesn't refer to them as *requirements* because it's such a nerdy word. She calls them *literals* because the reader read and literally interpreted what was written. Looking at the photos, I can imagine someone listening to a customer and writing down what he wants, then handing that to someone else who'll decorate a cake.

Customer: Hello, I'd like to order a cake.

Employee: Sure, what would you like written on it?

Customer: Could you write "So long, Alicia" in purple?

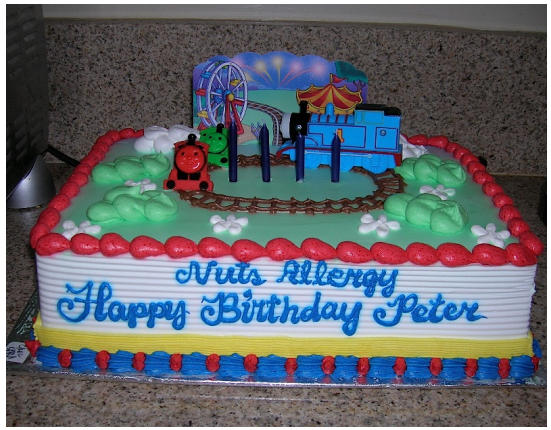
Employee: Sure.

Customer: And put stars around it?

Employee: No problem. I've written this up, and will hand it to my cake decorator right away. We'll have it for you in the morning.

This is the result:

Here's another. In software development, we call these *nonfunctional requirements*:



These are funny examples, and we can laugh about wasting twenty bucks on a cake. But sometimes the stakes are much greater than that.

You've probably heard the story about the 1999 crash of a $125 million NASA Mars Climate Orbiter.[1] OK, maybe you haven't. But here's the punch line. If any project is sunk up to its eyeballs in requirements and written documentation, it's a NASA project. However, despite all the filing cabinets full of requirements and documentation, the orbiter crashed because while NASA used the metric system for its measurements, members of the Lockheed Martin engineering team used the old imperial measurement system to develop navigation commands for the vehicle's thrusters. While no one knows exactly where the orbiter ended up, some think it has found its happy place orbiting the sun somewhere past Mars.

Ironically, we put stuff in writing to communicate more clearly and to avoid risk of misunderstanding. But, way too often, the opposite is true.

> *Shared documents aren't shared understanding.*

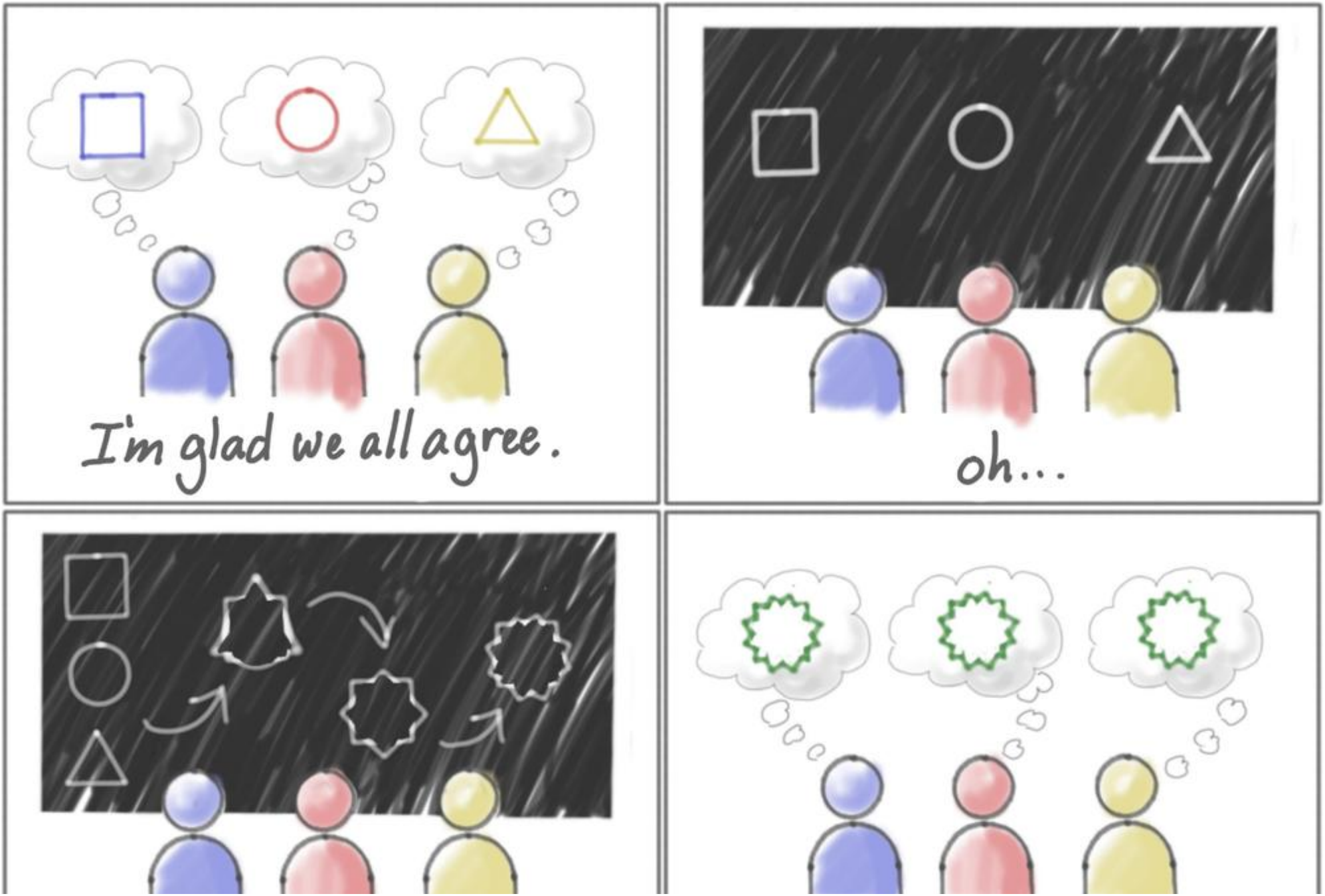
Stop for a minute and write that down. Write it on a sticky note and put it in your pocket. Consider getting it tattooed somewhere on your body so you can see it when you're getting ready for work in the morning. When you read it, it'll help you remember the stories I'm telling you now.

Shared understanding is when we both understand what the other person is imagining and why. Obviously, there wasn't shared understanding between several cake decorators and the people who gave them instructions in writing. And, at NASA, someone important didn't share understanding with others working on the guidance system. I'm sure if you've been involved in software development for a while, you don't have to reach back far in your memory to recall a situation where two people believed they were in agreement on a feature they wanted to add to the software, but later found out that the way one imagined it was wildly different from the other.

Building Shared Understanding Is Disruptively Simple

A former coworker of mine, Luke Barrett, first drew this cartoon to describe this problem. I asked him where he first saw it, but he didn't remember. So someone out there isn't getting the credit he or she deserves. For years I saw Luke step through these four frames as slides in a PowerPoint deck while I casually dismissed

them as interesting but obvious. Apparently I've got a thick head. It's taken me many years to understand how this cartoon illustrates the most important thing about using stories in software development.



The idea is that if I have an idea in my head and I describe it in writing, when *you* read that document, you might quite possibly imagine something different. We could even ask everyone, “Do you all agree with what’s written there?” and we might all say, “Yes! Yes, we do.”

However, if we get together and talk, you can tell me what you think and I can ask questions. The talking goes better if we can externalize our thinking by drawing pictures or organizing our ideas using index cards or sticky notes. If we give each other time to explain our thoughts with words and pictures, we build shared understanding. It’s at this point, though, that we realize that we all understood things differently. That sucks. But at least now we know.

It’s not that one person is right or wrong, but that we all see different and important aspects. Through combining and refining our different ideas, we end up with a common understanding that includes all our best ideas. That’s why externalizing our ideas is so important. We can redraw sketches or move sticky notes around, and the cool thing is that we’re really moving ideas around. What we’re really doing is evolving our shared understanding. That’s super-hard with just words alone.

When we leave this conversation, we may still name the same feature or enhancement, it’s just that now we actually mean the same thing. We feel aligned and confident we’re moving forward together. That’s the quality we’re managing to. And, sadly, it’s intangible. You can’t see or touch “shared understanding,” but you can feel it.

Stop Trying to Write Perfect Documents

There are a great number of people who believe that there’s some ideal way to document—that, when people read documents and come away with different understandings, it’s either the reader’s fault or the document writer’s. In reality, it’s neither.

The answer is just to stop it.

Stop trying to write the perfect document.

Go ahead and write something, anything. Then use productive conversations with words and pictures to build shared understanding.

> *The real goal of using stories is shared understanding.*

Stories in Agile development get their name from how they should be used, not what you write down. If you’re using stories in development and you’re not talking together using words and pictures, you’re doing it wrong.

If your goal in reading this book is to learn to write better stories, you’ve got the wrong goal.

Good Documents Are Like Vacation Photos

If I show you one of my vacation photos, you might see my kids on a beach and politely say, “That’s cute,” but when I look at my vacation photo, I remember a particular beach in Hawaii that we had to drive more than an hour on a deeply rutted four-wheel-drive trail, and then hike another half-hour over lava fields to get to. I remember my kids whining, saying nothing could possibly be worth this, and me wondering the same thing. But it was. We enjoyed a blissful day on an incredible beach where very few people were, which is why we took the trouble to get there. The turtles came up on the shore to bask on the sand, which was the icing on the cake of this fabulous day.

Of course, if you look at the picture you won't know all that because you weren't there. I remember all that because I was.

For better or worse, this is the way documents actually work.

If you participate in lots of discussions about what software to build, and then create a document to make sense of it, you might share it with someone else who was there. You might both agree it's good. But remember, your shared understanding is filling in details that aren't in the document. Another reader who wasn't there won't get the same things from it that you will. Even if she says she gets it, don't believe her. Get together and use the document to tell a story the same way I used my vacation photo to tell you my story.

Document to Help Remember

I've heard people joke, "We're using an Agile process because we've stopped writing documents." It's a joke for people who know, because a story-driven process needs lots of documents to work. But those documents don't always look at all like traditional requirements documents.

It takes talking and sketching and writing and working with sticky notes or index cards. It's pointing to documents we brought into the conversation and marking them up with highlighter and scribbled notes. It's interactive and high energy. If you're sitting at a conference table while a single person types what you say into a story management system, you're probably doing it wrong.

When you're telling stories, most anything can be used as a tool to communicate. And as we tell these stories, and write lots of notes, and draw lots of pictures, we need to keep them. We carry them around to look at later, photograph, and retype into more documents.

But, remember, what's most important isn't what's written down—it's what we remember when we read it. That's the vacation photo factor.

Talk, sketch, write, use sticky notes and cards, and then photograph your results. Even better, shoot a short video of you talking through what's on the board. You'll remember lots of details in a remarkable depth that you could never possibly document.

To help remember, photograph and shoot short videos of the results of your conversations.

Talking About the Right Thing

There are lots of people who believe their job is collecting and communicating requirements. But it's not.

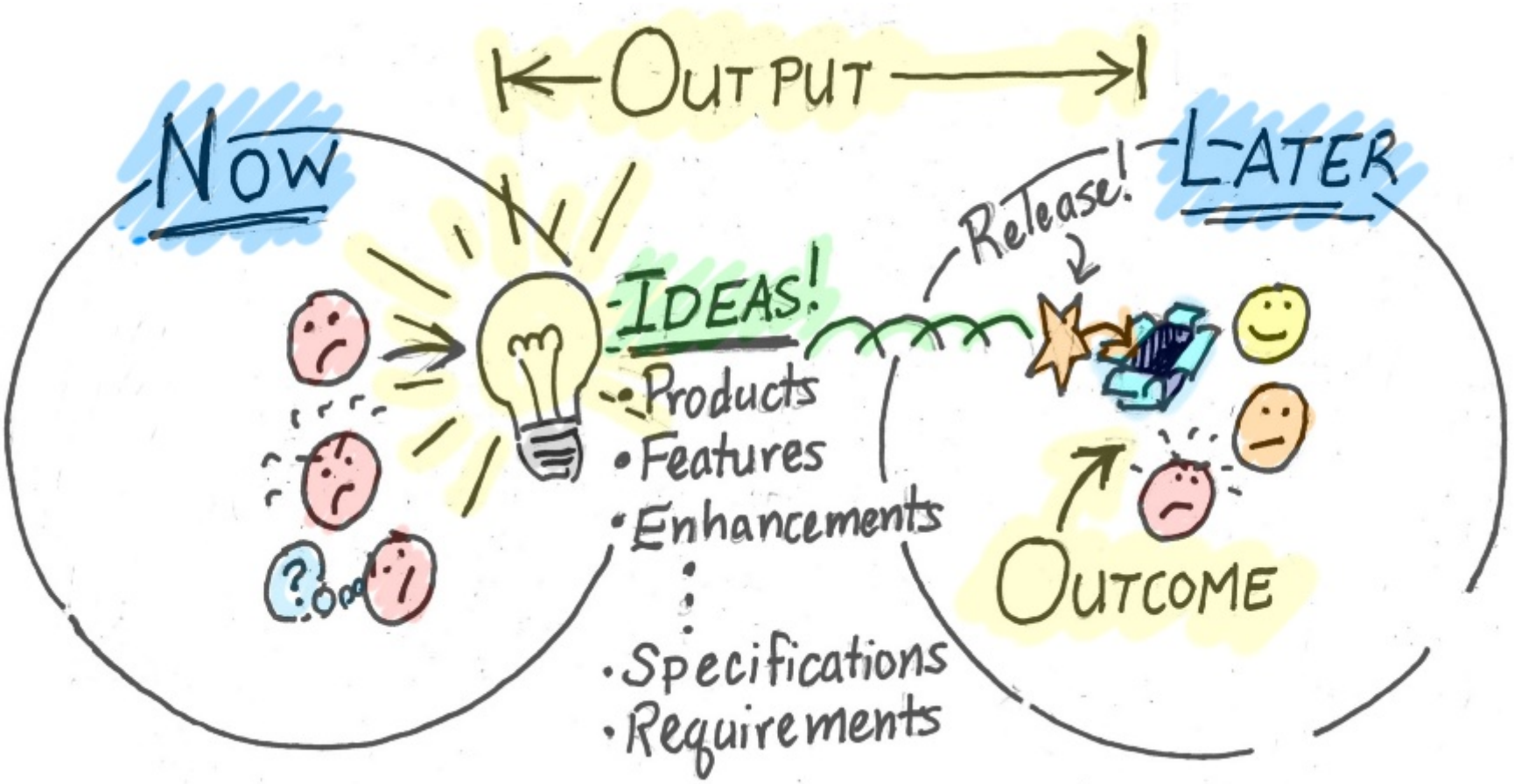
> *The truth is, your job is to change the world.*

Yes, I said that to get your attention. And, yes, I know that sounds like hyperbole. That's because that phrase is usually associated with world peace, eliminating poverty, or even more far-fetched goals like getting politicians to agree with one another. But I'm serious. Every great idea you turn into a product solution changes the world in some small, or not-so-small, way for the people who use it. In fact, if it doesn't, you've failed.

Now and Later

There's a simple, change-the-world model that I personally use and keep in my head all the time, and you need to keep it in your head too while you're having story conversation and building shared understanding.

I draw the model like this:



The model starts by looking at the world as it is now. When you look at the world as it is now, you're going to find people who are unhappy, mad, confused, or frustrated. Now, the world's a big place, so we'll focus mostly on the people who use the software we make, or the people we hope will use it. When you take a look at what they're doing—and the tools they use and how they're doing things—you're going to come up with ideas, and the ideas might be for:

- Entirely new products you can build
- Features to add to an existing product
- Enhancements to products that you've built

At some point in time, you'll have to communicate details about your ideas to some other people, and you might start to do some design and specification. If you're going to hand all this off to someone else, then you might indeed call all these details your *requirements*. But it's important to remember that requirements are just another name for the ideas we have that would help people.

Given those requirements, we go through some process that results in a delivery, and out comes some software that actually lands in the world, and it lands in the world *later*. And what we hope is true is that those people who were initially unhappy, mad, frustrated, or confused will become happy when that software lands. Now, they're not happy because they saw the pretty box it came in—software doesn't usually come in boxes these days anyway. They're not happy because they read the release notes, or downloaded the app to their mobile device. They're happy because when they use the software, or the website, or the mobile app, or whatever you've built, they do things differently—and that's what makes them happy.

Now, the truth is that you can't please everyone all of the time. Your mother should have told you that. Some people will be happier than others with whatever it is that you produce, and some might still be unhappy no matter how hard you've worked and how amazing your product might be.

Software Isn't the Point

Everything between the idea and the delivery is called *output*. It's what we build. People working in Agile software development will deliberately measure *velocity* of output and try to speed up their rate of output. As people are building software, they are, of course, concerned about the cost of what they're doing and the speed at which what they're doing gets done, as they should be.

But, while it's necessary, the output isn't the real point; it's not the output that we really wanted. It's what comes after as a result of that. It's called *outcome*. Outcome is what happens when things come out—that's why it's called that—and it's difficult because we don't get to measure outcome until things do come out. And we don't measure outcome by the number of features delivered, or what people have the capability to now do. We measure what people actually do differently to reach their goals as a consequence of what you've built, and most important, whether you've made their lives better.[2]

That's it. You've changed the world.

You've put something in it that changes the way people can reach their goals, and when they use it, the world is different for them.

If you remember, your goal isn't to just build a new product or feature. When you have conversations about that feature, you'll talk about who it's for, what they do now, and how things will change for them later. That positive change later is really why they'd want it.

> *Good story conversations are about who and why, not just what.*

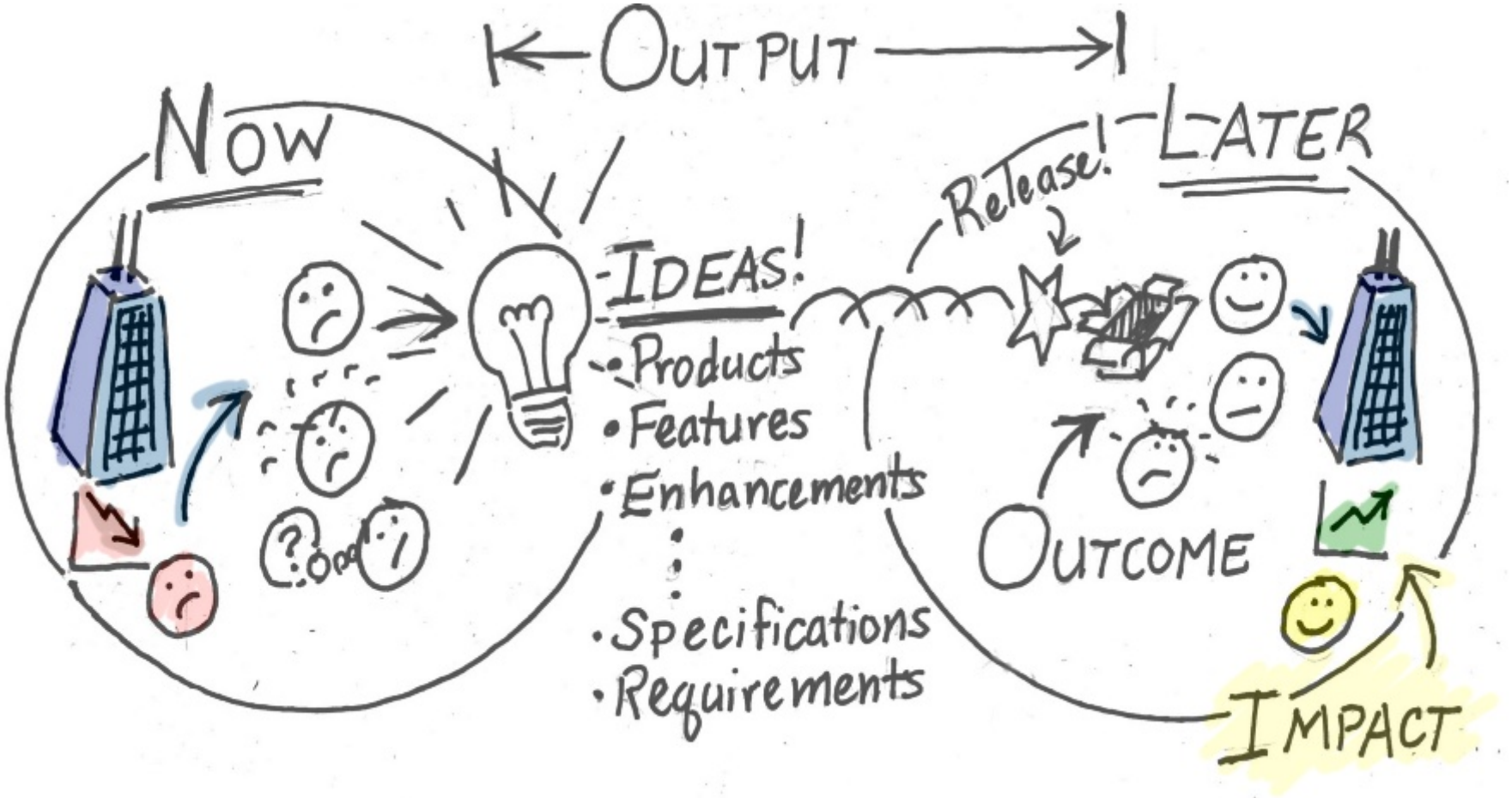
OK, It's Not Just About People

I care about people as much as the next guy, but truthfully, it's not just about making people happy. If you work for a company that pays you and others, you've got to focus on what ultimately helps your organization earn more, protect or expand its market, or operate more efficiently. Because, if your company isn't healthy, then you won't have the resources (or the job) to help anyone.

So I've got to revise this model a bit. It actually starts by looking inside your organization. There you'll find even more people who aren't happy. And it's usually because the business isn't performing as well as they'd hope. To fix this, they may have ideas to focus on specific customers or users and to make or improve the software products they're using. You see, it ultimately is about people, because:

Your company can't get what it wants unless your customers and users get something they want.

The flow continues by choosing the people to focus on, the problems to solve, and the ideas to turn into working software. And from there—if the customers buy, and the users use it, and people are happy—eventually the business that sponsored this development will see the benefit it's looking for. That'll be reflected in things like increased revenue, lower operational costs, happier customers, or expanded market share. This makes lots of people inside your company happy. It should make you happy, too, since you've just helped your company stay healthy while making real people's lives better in the process. It's a win-win.



It's that longer-term stuff that happens as a consequence of good outcomes that's I'll label *impact*. Outcomes are often something you can observe right away after delivery. But impact takes longer.

Build Less

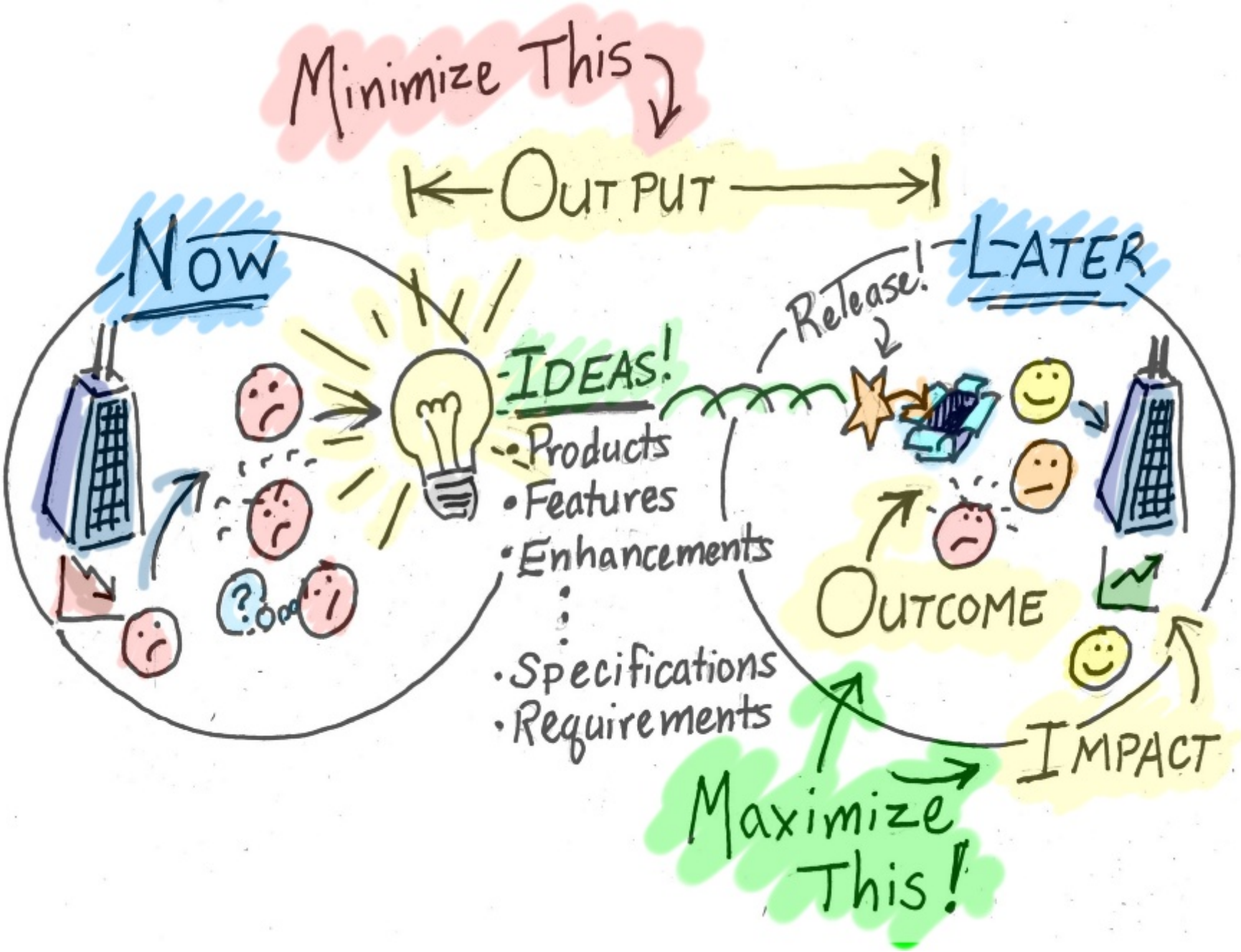
There’s an uncomfortable truth about the software world, and I suspect it’s true of lots of other places. But I know software. And what I know is that:

There’s always more to build than we have time or resources to build—always.

One of the common misconceptions in software development is that we’re trying to get more output faster. Because it would make sense that if there was too much to do, doing it faster would help, right? But if you get the game right, you will realize that your job is not to build *more*—it’s to build *less*.

Minimize output, and maximize outcome and impact.

At the end of the day, your job is to *minimize* output, and *maximize* outcome and impact. The trick is that you’ve got to pay close attention to the people whose problems you’re trying to solve. These include the people who will choose to buy the software to solve a problem in their organizations, the *choosers*, as well as the people who use it, the *users*. Sometimes they’re the same people. Sometimes they’re not.



Your business has lots of possible users and customers it could focus on. Your businesses strategy should give you some guidance about who to focus on to get the impact you want. I promise you that no business has the resources to make *everyone* happy—it's just not possible.

Don’t get me wrong here. Building more software faster is always a good idea. But it’s never the solution.

More on the Dreaded “R” Word

For almost the entire first decade of my software career, which I spent building software for brick-and-mortar retailers, I got away without using the word *requirements*—at least, not much. It just wasn’t a relevant term for what I was doing. I had lots of different customers who all had specific ideas about what would help them. I also knew I worked for a company that had to make money by selling my product. In fact, I’d spent long hours standing at trade shows helping my company sell its product to a wide variety of customers. I knew at the end of the day that I would have to continue to work with those customers after I shipped the products my team and I developed, and so I diligently worked to act in their best interest. This meant I couldn’t give everyone everything they wanted, because they wanted different things. And my company and team didn’t have infinite time, so I had to work hard to figure out the least I could build to make people happy. That may sound frustrating, but it’s actually the fun part.

As the company grew, we added more traditional software people. At one point, the head of a different team came to me and said, “Jeff, I need you to make these changes to the product you’re working on.”

I said, “Great, no problem. Tell me who they’re for and what problems this solves for them.”

Her response? “They’re the requirements.”

I replied, “I get it. Just tell me a bit about who they’re for, and how they’re going to use this, and where it fits into the way they work.”

She looked at me like I was stupid and said to me one last time with an air of finality, “They’re *requirements*.”

It was at that moment that I learned that the word *requirements* actually means *shut up*.

For a great many people, that’s exactly what requirements do. They stop conversations about people and the problems we’re solving. The truth is, if you build a fraction of what’s required you can still make people very happy.[3]

Remember: at the end of the day, your job isn’t to get the requirements right—your job is to change the world.

That’s All There Is to It

If you get nothing else from this book, remember these things:

- Stories aren’t a written form of requirements; telling stories through collaboration with words and pictures is a mechanism that builds shared understanding.
- Stories aren’t the requirements; they’re discussions about solving problems for our organization, our customers, and our users that lead to agreements on what to build.
- Your job isn’t to build more software faster: it’s to maximize the outcome and impact you get from what you choose to build.

Stories as they're intended are a completely different way of thinking about the challenges we face working together to create software—and lots of other things, for that matter. If you can work together effectively and create things that solve problems, you will rule the world. Or at least some small part of it inhabited by your products.

As you read this book, my hope is that you get back to the basics of using stories. I hope you work together with others, telling stories about your users and customers and how you can help them. I hope you draw pictures, and build big sticky-note models. I hope you feel engaged and creative. I hope you feel like you're making a difference. Because when you do it right, you are. And it's a lot more fun, too.

Now it's time to talk about the most fun you can possibly have telling stories, and that's when you're using a story map.

---

[1] There are a lot of articles that try to describe what went wrong with the Mars Orbiter. Here's one of them: http://www.cnn.com/TECH/space/9909/30/mars.metric.02/.

[2] The clean language and distinction between *output* and *outcome* was first made clear to me in a talk by Robert Fabricant called "Behavior Is Our Medium". Prior to that, I'd struggled with language that was clear in my head—and everyone else's too. Happily, it was clear in Robert's head.

[3] Because I strongly agree with the sentiment, I'm paraphrasing the way Kent Beck cautions against the misuse of the term requirement in his book *Extreme Programming Explained* (Addison-Wesley).