CHAPTER 1

The Big Picture

"I love Agile development! Every few weeks we see more working software. But it feels like I've lost the big picture."

If I had a dime for every time I heard something like that from an Agile team member, I'd have…well…a lot of dimes. I hear it a lot. You may have even said something like that yourself. Well, I've got good news for you. Using an Agile process and a story-driven approach doesn't mean you have to sacrifice the big picture. You can still have healthy discussions about your whole product and still see progress every few weeks.

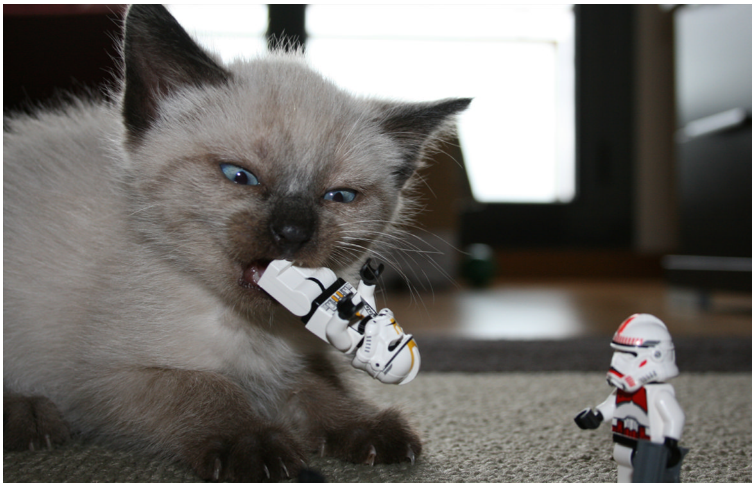
Since you've patiently read the "Read This First" chapter, I'm going to bypass all the junk about stories and proceed directly to how story maps solve one of the biggest problems in Agile development. If you're already familiar with writing stories on Agile projects, this chapter may be enough to get you started.

The "A" Word

If you're reading this book, you likely know that story mapping is a way to work with user stories as they're used in Agile processes. Now, it's at this point that every other book that has something to do with Agile development reproduces the "Manifesto for Agile Software Development," that thing written in 2001 by 17 guys who were frustrated with some of the big counterproductive process trends going on at the time. I'm glad they wrote it. And I'm glad that the impact of their work has been felt by so many.

But I'm sorry to disappoint you—I'm not going to reprint the manifesto and gush about why it matters. I believe you already know why it does. And, if you haven't read the manifesto, then you should.

In the space that the manifesto would have taken up in this chapter, I am instead including a funny kitten photo.[1] Why? Because it has been proven time and time again that funny kitten photos on the Internet get far more attention than *any* manifesto could ever hope to.



So, you might wonder, what does this kitten have to do with Agile? Actually, nothing. But Agile definitely has something to do with this book, and with stories and the evolution of story mapping.

<Cue the flashback music…>

I was working at a startup in San Francisco in 2000, and the company had hired Kent Beck (the guy who created Extreme Programming and first described the idea of stories) as a consultant to get the software development process going. I'm rewinding way back, but the important thing is this story idea is an old one. If you're just starting out with using stories, you lost any early adopter status you could have had a decade or so ago. Kent and others who pioneered Extreme Programming knew that all those ways of doing requirements in the past didn't work out well. Kent's simple idea was that we should get together and

1. Photo taken by Piutus, found on Flickr and licensed under the Creative Common Attribution license.

tell our stories; that by talking we could build shared understanding, and together we'd arrive at better solutions.

Telling Stories, Not Writing Stories

When I first heard the term *story*, it bugged me. I'll admit it. The idea that we'd trivialize the important things that people wanted by calling them stories didn't seem right. But I'm a slow learner—a point I brought up earlier when discussing shared understanding. It took me a while to really get that:

Stories get their name from how they should be used, not what should be written.

Even before I'd really understood why stories had that name, I realized that I could write down a bunch of stories—a sentence or a short title —on sticky notes or cards. I could move them around and prioritize them to decide which one was more important. Once I decided that one was more important than another, then we could start having a discussion about it. This was super-cool. Why hadn't I ever written things on cards and organized them this way before?

The problem was that this one card could be something that might take a software developer just a couple hours to add to a product, or maybe a couple days or a couple weeks, or maybe a month—who knew? I didn't—at least not until we started talking about it.

I got into a nasty argument while working with stories on my very first Agile project when I began a story conversation and learned that my story was too big. I'd hoped to get this story done in the next iteration. The developers I spoke with informed me otherwise. I felt like I'd done something wrong. The developers identified a small part we could talk about that could be accomplished in our next iteration. But I left frustrated that we couldn't talk about the big picture. I really wanted to understand how much time the big thing I really needed would take. I'd hoped this discussion would accomplish that, and it didn't.

Telling the Whole Story

In 2001 I left the team I was on and started doing things differently. I, and my team, tried an approach to writing stories that focused on the big picture. We worked to understand the product we were building and to make tradeoffs together. We used that bunch of index cards with story titles to organize our thoughts and break down that big picture into the small parts we could build next. In 2004, I wrote my first article about this idea. I didn't coin the term *story mapping*, however, until 2007.

It turns out that the name you give something matters. It was after giving the practice a good name that I really saw it spread. I thought it was a great invention at the time—that is, until I started running into more people who were doing similar if not exactly the same things. I'd discovered a *pattern*.

I first heard this definition of a pattern from my friend Linda Rising: when you tell someone about a great idea and he says, "Yeah, we do something like that, too." It's not an invention, it's a pattern.

Story mapping is a pattern. It's what sensible people do to make sense of a whole product or whole feature. It's what they do to break down large stories into smaller ones. Don't feel bad if you didn't arrive at it on your own. You would have eventually. But reading this book will save you weeks or months of frustration.

Story maps are for breaking down big stories as you tell them.

Today, company after company has adopted the idea of story mapping. My friend Martina at SAP said in a message she sent in September 2013 that:

> …at this point more than 120 USM [User Story Mapping] workshops have officially been recorded. A lot of POs just simply love it! It is simply a well-established approach at SAP.

Every week I hear from someone else from somewhere else telling me how mapping stories helped solve a problem for them. These days, I learn more from talking to others than I ever could on my own.

The original idea of stories was a simple one. It turned our focus away from shared documents and toward shared understanding. A common way to use stories is to build a list of them, prioritize them, and begin talking about them and then turning them into software one at a time. That sounds pretty reasonable when you hear about it. But it can create some big problems.

Gary and the Tragedy of the Flat Backlog

A few years ago I met Gary Levitt. Gary was a businessperson in the process of launching a new web product. The web product is out there right now, and it's called Mad Mimi, which when Gary conceived of his product, was short for *music industry marketing interface*.[2] Gary is a musician who had his own band. He managed his band, helped manage others, and was also a studio musician and created recordings for clients.

The day I met Gary he had an order from the Oprah Winfrey show for dozens of intros and outros, little bits of music that are used to go out to and come in from commercials and things like that. Producers of television shows buy those the way people laying out a newsletter buy clip art, so it's like audio clip art. Gary had an idea for a fairly big application that would help musicians like him and people he knew to collaborate with one another on projects like the one he was working on, along with lots of other things a band manager and musician would need to do to manage and promote his band.

Gary wanted to get the software built so he worked with somebody, and that somebody was working in an Agile way. That person told Gary to write down a list of all the things he wanted, prioritize the list, and then they would talk about the highest-valued things—the most important—and start building them one at a time. That list of things

2. Read about Gary in the *Business Insider* article "How This Guy Launched A Multi-Million Dollar Startup Without Any VC Money".

is what Agile processes refer to as a *backlog*, and it seemed to make sense to Gary to create the list and start with the most important things first. So that's what he did.

Gary created his backlog and the development team started building things a bit at a time. In the meantime, Gary was hemorrhaging cash as he continued to pay for each piece of software that was built. The software was slowly taking shape, but Gary could tell it was going to take a lot longer for it to match his vision and he was going to run out of cash long before then.

I knew the person who was working with Gary. My friend knew Gary was stressing out and wanted to help him. The somebody I knew asked if I could have a conversation with Gary, to talk with him and help him get his ideas organized. I contacted Gary and made arrangements to meet him at his office in Manhattan.

Talk and Doc

Gary and I started talking. And as he talked, I wrote cards with key points from what he said. There's a mantra that I like when I build story maps. I'll say "talk and doc" (short for the verb *document*), which basically means don't let your words vaporize. Write them down on cards so you can refer back to them later. You'll notice how pointing to a few words on a card quickly helps everyone recall the conversation about it. We can slide them around the table where we can reorganize them. We start using useful words like *this* and *that* as we point to cards. It saves lots of time. Helping Gary externalize his thoughts was critical to getting shared understanding. It wasn't a habit for him, so it was easy for me to write the cards as he told the story.

Talk and doc: write cards or sticky notes to externalize your thinking as you tell stories.

We started by placing cards on a tabletop, but quickly ran out of space. Gary was moving offices the day I visited with him, and much of the furniture in the New York City loft where he was located was off the floor. So we moved our growing map of cards onto the floor.

At the end of the day, the floor looked like this:

Think — Write — Explain — Place

When working with a team to build a story map, or having discussions about anything, create a simple visualization to support your discussion. One of the things that goes wrong is lots of ideas *vaporize*—that is, we say them, and people nod as if they've heard. The ideas are not written down or referred to. Then, later in the conversation, the ideas come up again and unfortunately need to be re-explained because people didn't really hear or forgot them.

Get in the habit of writing down a little about your idea before explaining it.

1. If you're using cards or sticky notes, *write* down a few words about your idea immediately *after thinking* it.
2. *Explain* your idea to others as you point to the sticky note or card. Use big gestures. Draw more pictures. Tell stories.
3. *Place* the card or sticky into a shared workspace where everyone can see, point to, add to, and move it around. Hopefully, there will be lots of other ideas from you and others in this growing pile.

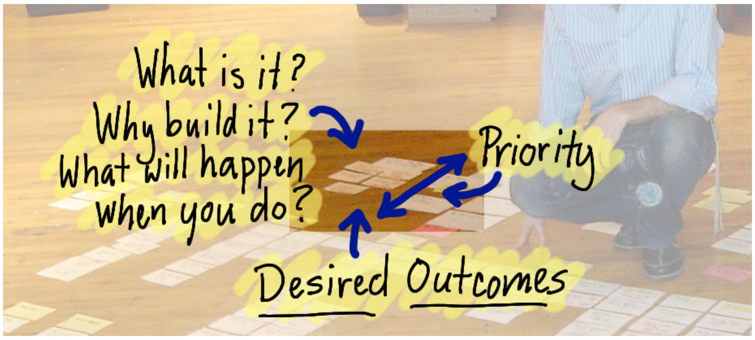
I find that when I'm doing my best to listen to others, what they're saying causes me to think of other ideas. I used to try to hold those ideas in my head and wait for a moment to inject them into the conversation, resorting to outright interruption if the time didn't come soon enough. But then I realized I'd stopped listening to the person who was talking, as my limited brainpower was focused on recalling my great idea. Today, I simply scribble the idea on a sticky note and set it aside to wait for a better point in the conversation to inject it. Somehow writing it pops it out of my head so I can focus on what I'm hearing. And reading it from the sticky later helps me recall my idea and explain it.

I wasn't here to capture Gary's requirements. And the first thing we talked about wasn't that list of features. We had to back up a bit and start at the beginning.

Frame Your Idea

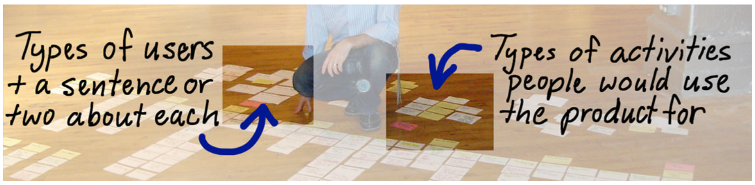
Our first conversation focused on framing his product idea. We talked about his business and what his goals were. *Why are you building this? Tell me about the benefits for you and for the people who will use this. What problems does it solve for those people and for you?* As you read this you might detect I've got that now-and-later model in my head. I'm trying to understand the outcomes Gary is looking for, not the output he wants to build.

If I put two cards down, one above the other, then people assume that the one above is more important. Without saying a word, if I simply slide a card above another, I've indicated something about importance. Try that with a list of goals. Purposely put them in the wrong order and watch the person you're working with reach out to adjust them. I did this with Gary and his goals, and it helped him express what was more important to him.



Describe Your Customers and Users

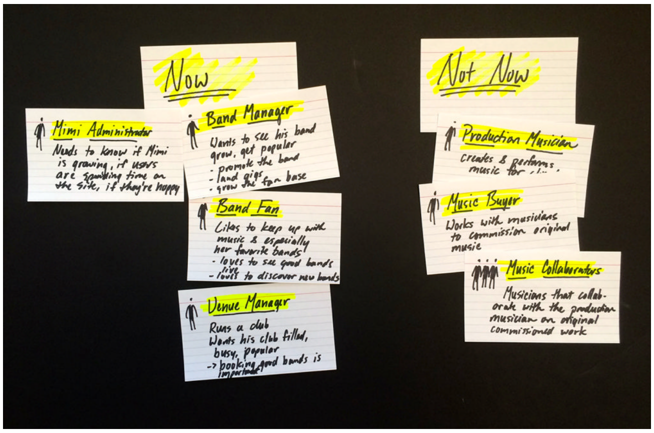
Gary and I continued to talk and doc. The next conversation Gary and I had was about the customers who would buy, and users who would use, his piece of software. We listed the different types of users. We talked about what benefits they would get, and asked why they would use the product and what we thought they would do with it. What was in it for them? We built a big pile of those. The cards naturally seemed to fall with most important users higher in the pile. Funny how it works out that way without an explicit decision.


Types of users + a sentence or two about each
Types of activities people would use the product for

Before we'd gone into any detail at all, I could already see that Gary's vision was big. One of the tough realities about software development is that there's always more to build than we have time and money for. So the goal should *never* be to build it all. The goal is to minimize the amount we build. So the first question I asked Gary was, "Of all these different users and the things they want to do, if we were to focus on thrilling just one of those users, who would it be?"

Gary chose one and we started to really tell stories.

Mad Mimi User Types


Now
Mimi Administrator
Needs to know if Mimi is growing, if users are spending time on the site, if they're happy
Band Manager
Wants to see his band grow, get popular
- promote the band
- land gigs
- grow the fan base
Band Fan
Likes to keep up with music & especially her favorite bands
- loves to see good bands live
- loves to discover new bands
Venue Manager
Runs a club
Wants his club filled, busy, popular
-> booking good bands is important
Not Now
Production Musician
Creates & performs music for
Music Buyer
Works with musicians to commission original music
Music Collaborators
Musicians that collaborate with the production musician on original commissioned work

These are the different types of users Gary described for Mad Mimi. Just naming them and writing a little about what they want helped us both see that there was a lot here. Even before discussing features, we'd decided to defer creating software for some types of users.

Tell Your Users' Stories

I next said, "OK, let's imagine the future. Let's assume for a minute this product is live and let's talk about a day in the life of someone who uses it and start telling the story. First, they would do this, and then this, and so on and so on." And we told the story in a flow from left to right. Sometimes we backtracked and put things to the left of other things, and because they were on cards, we could easily rearrange them.

The other interesting thing that happens naturally when working with cards is if I put one to the left and another to the right, without saying a word I've indicated sequence. This is kind of magical for me—but I'm easily entertained. I marvel at how much we can communicate without saying a word.

Reorganizing cards together allows you to communicate without saying a word.

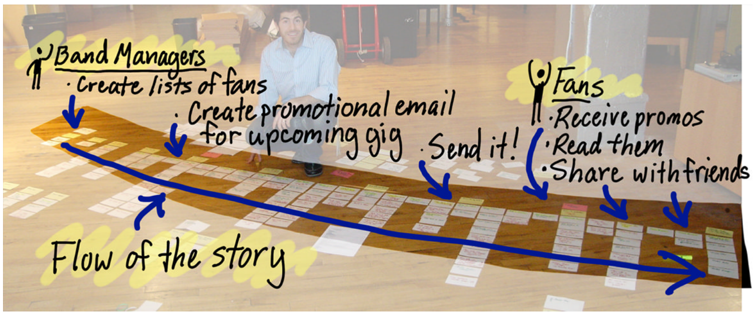
As we talk and doc, and as I write down our conversation, we're building something really important. No, it's not that pile of cards on the floor. The something that's really important is *shared understanding*. We're getting on the same page. This is something Gary had never done with anyone before about his product idea, at least not at this level of detail. He'd never even given it this much thought himself. The high points were in his head, sort of like the action scenes you'd see in a movie preview.

Before now, Gary had done what he was asked to do. He'd written a bunch of story titles, put them in a list, and talked about them one at a time. The conversations were more about the details of what to build and less about this big picture. And there were a lot of holes in Gary's big picture. You'll find that no matter how clear you are about your story, talking through it while you map will help you discover the holes in your own thinking.

Mapping your story helps you find holes in your thinking.

As we dug deeper, we realized that the story also wasn't just about one user. Gary's started with a band manager who wanted to promote his band and the work he was doing to create the promotion and email it to fans. Then we quickly had to talk about the fan of the band, and tell her story about seeing the promotion and then making plans to see a show.

Then, if we were promoting the band someplace, we'd need to tell the story of the venue's manager and the information he'd like to learn about the promotion. By this time, our map was wide enough that we bumped into the wall, so we had to continue the story in another layer below the first. That's why the map in the photograph has two layers.



During the story, sometimes Gary would get to a part where he was excited and he'd start describing lots of details. One card above another can indicate priority. But it can also mean *decomposition*, which is just a fancy word for smaller details that are part of a bigger thing. As Gary described the details, I'd record them on a card, and place them below the big user step above. For instance, when Gary described creating the flyer that band managers would use to promote their gigs, he was extra passionate and had lots of details to discuss.

Gary lived in New York City, and when bands are composing flyers he's imagining all these really cool things he sees stuck on walls and lampposts in New York. They might look like they were put together with glue and tape and then photocopied, but some were really elegant and artistic. After recording a handful of details, I said, "Let's come back and get to the details later. Let's continue on and move this story forward." It's easy to get lost in the details, especially the ones you're passionate about. But, when we're trying to get the big picture, it's important to get to the end of the story before catching all those details. Another mantra I use when mapping, at least at this stage, is "think mile wide, inch deep"—or for people in sane countries using the metric system: "kilometer wide, centimeter deep." Get to the end of the story before getting lost in the details.

Focus on the breadth of the story before diving into the depth.

Eventually we *did* get to the end of Gary's story. The band manager had successfully promoted a gig to thousands of fans who spread the word, and the show was a wild success. The product vision so far was clear in both our heads. I said, "Now let's go back and fill in the details and consider some of the alternatives."

Mimi's Big Story

If you read across the top of Gary's map, you'll see big activities like:

- Signing up
- Changing my service
- Viewing my band stats
- Working with my show calendar
- Working with my audience
- Publicizing a show
- Signing up for a band's email list
- Viewing promotions online

There were lots of other big things at the top of the map, but that's a good subset to give you an idea of what you'd write on a card. Notice how we can assume who does what. When Gary said, "Publicizing a show," he knew he was talking about the band manager. When I said, "signing up for the band's email list," Gary knew I was talking about the band's fan. Those cards were close by and easy to point to during our conversation.

"Publicizing a show" was a big thing. It broke down into these steps arranged left to right underneath the "Publicizing the show" card.

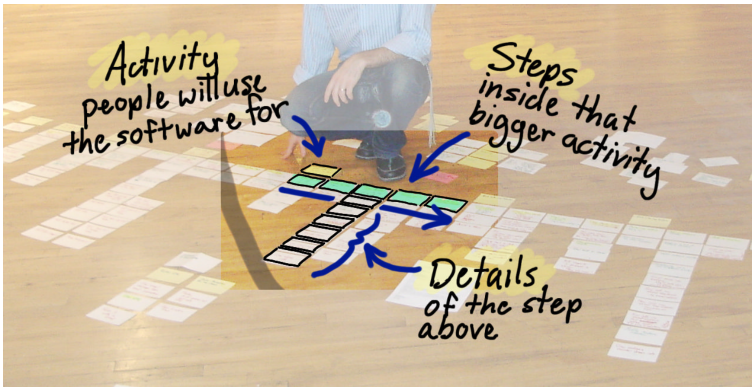
- Start a show promotion.
- Review the promo flyer Mimi created for me.
- Customize the promo flyer.
- Preview the promo flyer I created.

Notice how what we wrote on every card are short verb phrases that say what the specific type of user wants to do. Writing them this way helped us tell the story: "the band manager would then publicize the show. To do that he'd start a promotion, then review the flyer Mimi created, then customize it, and then…" Notice how when you put "and then" in between what's written on each card, you get a nice story.

Explore Details and Options

After we've got the breadth of the story map in place, it starts thickening up. The cards we put at the top of each of the columns in the map become big things, and then the details break down below them. We stop at each step in the user's story and ask:

- What are the specific things they'd do here?
- What are alternative things they could do?
- What would make it really cool?
- What about when things go wrong?


Activity people will use the software for
Steps inside that bigger activity
Details of the step above

At the end of this we'd gone back and filled in a lot of details. The result was that we had told the story about a day in the life of a band manager, as well as the other people important to the band manager's success: fans and venue managers.

The Details

If you look inside a story step like "Customize the promo flyer," you'd see details like:

- Upload an image
- Attach an audio file
- Embed a video
- Add free text
- Change the layout
- Start with a promotion I've used before

You can see that even these smaller steps will need a lot more discussion to work out the details. But at least we could begin to name them all.

Notice how what's written on the cards are also those short verb phrases that help you tell stories. We can string this together with phrases like "or he might" like this: "to customize the flyer the band manager might upload an image, or he might attach an audio file or embed a video, or…" It's pretty cool, really.

I asked Gary, "Now what? We have all these other users with other things they want to do—do you want to talk about them? You can see that if we keep talking we'll need a bigger room. And, Gary, if you do all this stuff, it will take a lot of money to build this product. We could talk about the rest of this stuff, but if we built this much and you launched your product and just did this, that looks like it'd be a valuable product."

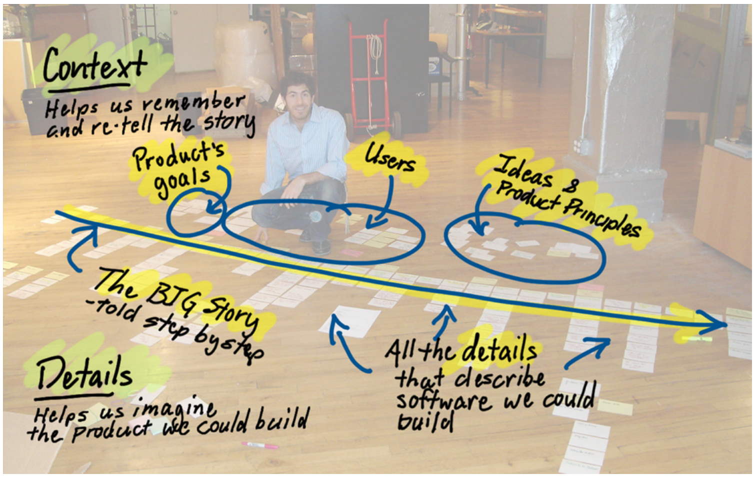
Gary agreed, and he said, "I'm going to stop there."

The sad part of this story is that I asked Gary, "You've been building a lot of software so far, but how much of the software you've built is on this map we've created?"

"Nearly none of it," Gary replied, "because when I built a list and prioritized things, I sort of assumed we needed to start somewhere else. I was thinking about the whole big vision of this thing, the vision that would have taken me years to reach, and now that we've had this discussion I wouldn't have started there at all."

Story mapping is all about having a good old-fashioned conversation and then organizing it in the form of a map. The part that most people look at is the map—that left-to-right shape with the steps people take to tell a big story. The top to bottom is about the details. But the critical parts that frame the product and give more context are often hung above and around the map. They're the product's goals, and information about its customers and users. If you keep a map on the wall, you'll find it's good idea to add user interface (UI) sketches and other notes around the map.

In just a day working together, Gary and I built shared understanding around the product he wanted to build. But there was a storm cloud forming above our heads, and we knew it. Inside each of those cards we wrote were lots of details and lots more discussions. And, for Gary, all those details and all those discussions equated to money he would need to spend to build software—money he didn't have. He'd learned one of the fundamental truths about software development: there's always more to build than you'll have time for.


Context
Helps us remember and re-tell the story
Product's goals
Users
Ideas & Product Principles
The BIG Story -told step by step
Details
Helps us imagine the product we could build
All the details that describe software we could build

Now, there are a lot of other big assumptions Gary was making about the people who'd use his product, and if they really wanted to, or really could use it as he envisioned. But, right now, those things weren't his biggest concern. He needed to work harder to minimize his product idea to something that was feasible first.

Gary's story eventually has a very happy ending. But in the next chapter I'll tell the story of another organization that learned it had way too much to build, and how it used a map to find a viable solution.

Artgility—Creativity in Art Meets Creativity in IT

Ceedee (Clare) Doyle, Agile Project Manager and Coach, Assurity Ltd, Wellington, NZ

Background

The Learning Connexion (TLC) is an art college in Wellington, New Zealand, that teaches art and creativity. TLC's programs are unique because they are based on "learning by doing"; that is, the practice is the theory. In conjunction with tutors, students develop briefs that connect with the ideas they choose to explore.

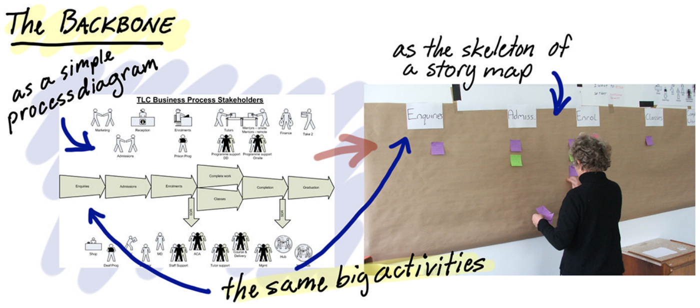
TLC was a typical small-to-medium-size organization that had developed ad hoc IT systems to support the needs it had at the time. Student information was collected in five different places and was different in each! TLC needed some way to manage students that would work for it and the way it teaches, which is quite different from most educational establishments.

TLC had no experience with IT projects. Each small application that had been built for it had been done by somebody's-brother's-friend's-flatmate's-dog using simple technologies like Microsoft Excel and Access. The sole commercial application (used for statutory reporting) double-handled data from the other four sources.

As a former student, I had kept in touch with the team and when they needed some help they contacted me. In 2009 I had been in IT for nine years and had wanted to do an Agile project for the last three, ever since I had heard about it. This was the right place, the right project, and the right time to do it!

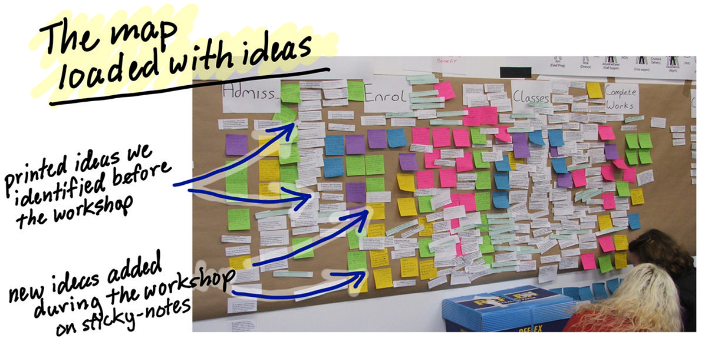
Project Phoenix

The initial workshops were going to be two half-day sessions with key staff members. I was working with a large, diverse group, and my goal was to develop shared understanding. I started with an overview of how story mapping works and an overview of the big steps in the school's student management process.



Up until I showed them this picture (the backbone of the story map), the team members each had an idea of what *they* did, but, as Alice the sponsor said, it was probably the first time they all had a clear picture of their own business process and how all the steps interacted.

From there we brainstormed what people wanted the system to do. The scope was *massive*, and the stories were *many*.


The map loaded with ideas
printed ideas we identified before the workshop
new ideas added during the workshop on sticky-notes
Admiss.
Enrol
Classes
Complete Works

The beauty was that these were creative people and they were used to the "appreciative enquiry" method, so braindumping everything they could think of that the system needed to do was something they took to like a baker makes bread.

The main headings (from the diagram) were Enquiries → Admissions → Enrollments → Classes → Complete work → Completion → Graduation.

Using the story mapping guidelines, we then walked through each section to make sure that it made sense, and got the flow for a student through each step of the process. Several people had lights go on suddenly, as they realized where they fit into the overall process and *why* they had to do some of their activities, and others realized they were being left out of certain steps that would make a big difference to them. Stepping through the story map and my emphasis on having stories vertically—which happened together—showed places where they could work together better and steps that were doubled up. Until this point, the team had little view of what everyone else was doing, but they suddenly developed shared understanding of how the whole process worked and a common lexicon. In one example, *Classes* was renamed *Delivery* because not all students attend classes.

When it came to prioritizing, it wouldn't have worked to identify the must-haves, should-haves, and could-haves—it was either in or out. It was very simple: "we cannot go live without this" above the line, and everything else below it. After we'd walked through the Enquiries, the team got it and spent the remainder of the day doing the rest. I was able to step out of it! The staff took over and started adding sub-headings to better describe that "these things all have to happen together, and then these things." So, by the end, they had created, as a group, a big picture of the steps a student follows to get from initial enquiry to graduation.

What started out as two half-day sessions turned into three full days of workshops in which people came and went as the need arose (they had to teach classes, and so forth). The flexibility of the workplace meant that nearly all the staff came through the Phoenix room and had their two cents' worth. They found the process really useful to get the big picture and for everyone to have their wants included. It also identified where there were gaps and made it easy to sift out what was really vital. At the end of it, we had a clear picture of what was to be in the first cut of the software.

CHAPTER 2

Plan to Build Less

There's always more to build than you have people, time, or money for. Always.

Speaking in absolutes like "always" and "never" always gets me in trouble. But with the statement above, I honestly can't recall a situation where it wasn't true although I have no sciencey data to back this up. No one has ever once come to me saying, "We were asked to add this new feature, and happily we had lots more time than we needed."

But one of the coolest things about using a story map is that it gives you and other collaborators a space to think through alternatives and to find a way to get a great outcome in the time that you have.

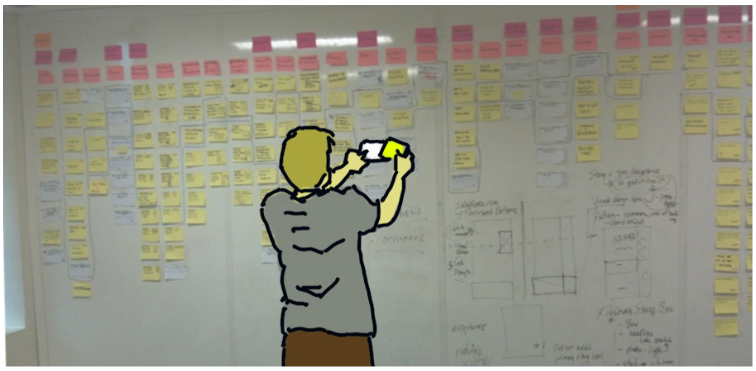
Grab a cup of coffee and settle into your chair. It's time for a story.

This story is about my friends at Globo.com, the largest media company in Brazil. Globo.com owns television and radio stations, produces made-for-TV movies and original programming, and publishes newspapers. It is a media monster in Brazil, and the largest Portuguese-language media company in the world.

Globo.com knows better than most organizations on the planet what unmovable deadlines really are. For example, it produces a cool version of a fantasy football game that's revised and improved every year for World Cup soccer—the sport that most of the planet refers to as football. If Globo.com is late with development of that game, it doesn't have the luxury of changing the release date. Why not? Because the rest of the world won't reschedule the World Cup. Globo.com will produce features and content for the Olympics games that Brazil will host in 2016, and I can guarantee you that it'll get it done in time—it has to. And it'll produce features and content for the release of numerous new television programs and reality TV shows. None of these things can be rescheduled if Globo.com is late. Globo.com must *always* finish on time. And, since that's the reality of its business, Globo.com is good at it. That's not because the company is faster than everyone else—sure, it's fast, but it's not *that* fast. It's because it's smart about doing less.

Mapping Helps Big Groups Build Shared Understanding

Take a look at this:



That's just a portion of a pretty big map built by leaders of eight teams from three different groups at Globo.com working together. The teams from Sports, News, and Entertainment built this together to think through the work they needed to do to rebuild, revamp, and renovate their underlying content management system. This is the system that drives all those news websites, sports websites, soap opera websites, features that help publicize and recruit people for reality TV shows, and much, much more. This massive system needs to be able to handle large quantities of video feeds, real-time scores and election results, photographs, fast-breaking news stories, and much more. It has a lot to do, and it needs to look good doing it.

When I arrived at the Globo.com offices the day they built this map, the teams working together were about to fall into the *flat backlog trap*. Individual teams had prepared their respective prioritized backlogs. It was already clear there was a huge amount of work to do, and each team depended on the other. For instance, to get a good news site out would need not just the news team, but also all the other teams that built the foundational components that let the news website use photos, videos, real-time data, and lots of other things.

I sat down with them and reminded them of something they already knew: "I understand that you're different teams because you're focusing on different areas, but it's a major revision of *one* content management system. You'll have to release together. You can't plan a release until you can see it all together. You've got to visualize all these dependencies." They agreed and quickly went to work reorganizing their individual backlogs into a map. Within a few hours they built a map on the wall using sticky notes that told the story of their content management system.

I wasn't in the room while team members worked together to build the map. But when I returned later in the day, I was amazed at how quickly they'd built it. They were pleased with themselves, and had every right to be. They'd made sense of several complex backlogs, organizing them into one coherent product story. And now each team could see where its work fit into the big picture.

Map for a product release across multiple teams to visualize dependencies.

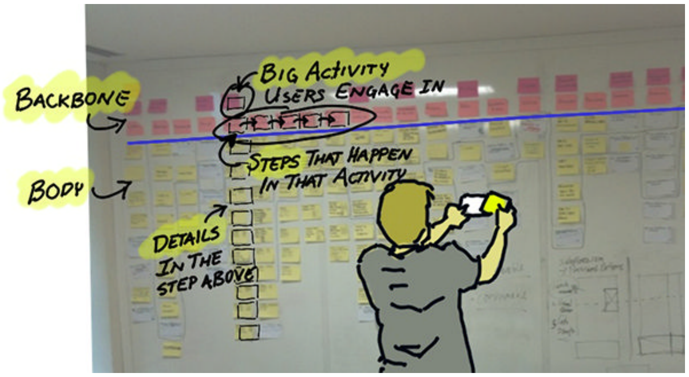
Anatomy of a Big Map

Globo's map is good example of what a typical map looks like *after* you've framed, mapped, and explored lots of details.

The Backbone Organizes the Map

At the top of the map is the *backbone*, which sometimes has a couple of different levels. You might start with the basic flow of the story, which is one level. But, when it gets really long, it's useful to go up one more level to summarize things further. Later, I'll add some language about what I like to put at each level, but I'm reminded of an old friend of mine who told me to stop trying to come up with precise language. "It's just big things and little things," he'd tell me. And he's right.

The whole thing kind of looks like you extracted the spine out of a weird, Seussian animal. You've got this long backbone at the top with lots of irregularly spaced vertebrae, and these long ribs of varying length hanging down from it.


BACKBONE
BIG ACTIVITY USERS ENGAGE IN
STEPS THAT HAPPEN IN THAT ACTIVITY
BODY
DETAILS IN THE STEP ABOVE

Map in Whole Deliverable Releases

This map was built by multiple teams at Globo. There are development teams responsible for video, and teams charged with the task of building the stuff on the backend that editors use to create and manage content. There are teams responsible for some of the underlying metadata and associations between data—that semantic markup junk that I can never quite grok. And there are people who handle the external presentation and how good this stuff looks when users and/or consumers see it. And still more people are looking after specific features relevant to news, sports, or entertainment.

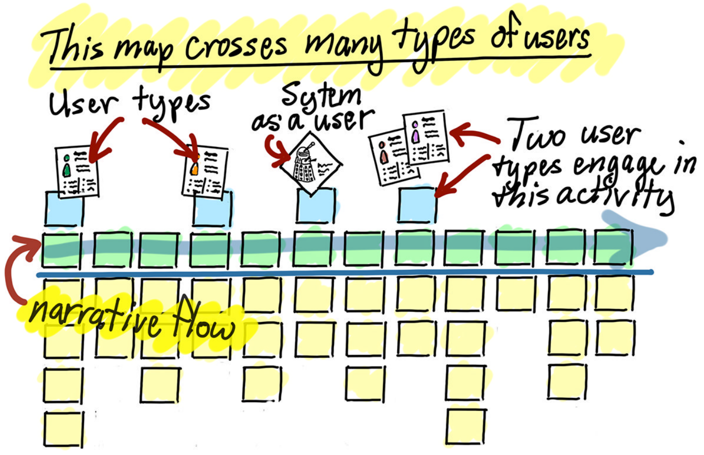
Multiple teams had to work together on this map because, for this major revision, no single team could release its part without the others. The teams built a single map because they needed to think through the release holistically.

Map in a Narrative Flow Across Many Users and Systems

The map started with people on the left, and the things they had to do to set up basic widgets for the screens that held news stories, pictures, and videos. There were other types of people who then combined those onto pages for soap opera or news websites. Then there were editors who added content to pages. This entire backbone tells the story of how lots of different people at Globo.com construct and manage content in its website.

When you read the backbone of the map from left to right, it tells a story about all the people who use the system and what they do in order to create and manage sites and content. The left-to-right order is what I call a *narrative flow*, which is an academic way of saying the order in which we'd tell the story. Of course, all these people are doing everything all at once, and sometimes things don't move in a perfect order, but we know that. We just put them in an order that helps us tell the story.

For this big system, that narrative flow has to *cut through many different users' and systems' stories*. I like to place stickies or simple persona thumbnails above the backbone so I can see who we're talking about at particular times in the story. And it's OK to anthropomorphize backend services or complex stuff the system does. My friends at SAP create fictitious personas for their systems and use pictures of R2D2 or C3PO from *Star Wars*.


This map crosses many types of users
User types
Sytem as a user
Two user types engage in this activity
narrative flow

Mapping Helps You Spot Holes in Your Story

When I talk with people who have built story maps, they'll tell me, "Every time we do this we find holes. We find things that we thought *another* team should be taking care of, but it didn't know. We find the necessary stuff in between the big important features that we'd forgot to talk about." By mapping together, Globo.com found some of that.

After you've envisioned the whole product or feature, it's easier to start to play the "What-About" game. That's where we start asking, "What about when this goes wrong?" Or "What about these other users?" Play What-About with any concern you have, and add stickies to the body of the map for feature ideas you'll need in order to address these things in your software. In Chapter 1, Gary played What-About to consider options and alternatives. When you do this with other teams, you'll find they're terrific at spotting problems that might arise where different systems connect to each other.

One of the criticisms people sometimes make about story mapping is that every time they sit down and create story maps, they end up with way too much. But it's my belief that we're just finding the stuff now that would have bitten us later on, and that's a good thing.

In the old-school approach to software development, when we'd find that new stuff later on—after we'd already estimated delivery time and committed to a delivery date—we'd call that new stuff *scope creep*. I personally believe that scope doesn't creep; understanding grows. And what happens when people build one of these story maps is that they find the holes in their understanding.

Scope doesn't creep; understanding grows.

There's Always Too Much

When I left the content management teams at Globo.com, everything was wonderful—understanding was rampant and the teams knew what to do. However, when I came back to check in with them a couple days later, they were struggling again because they realized that there was so much work to be done, that it would likely take more than a year to accomplish everything on the map. And, of course, as savvy readers can appreciate, when software developers say it's going to take a year to get something done, they really mean two years. It's not because they're incompetent, or that they are calendar-challenged, it's just that estimating the time to do something we've never done before is something we suck at. And, by nature, we're often optimistic animals.

They said to me, "There's way too much. We have a lot to do here and it's going to take a long time."

"Do you have to do it all?" I asked.

They, of course, replied, "Yes, because it's all part of one, big content management system."

"But projects don’t run that long around here," I replied. "I know your CEO and he’s going to want to see results much faster—right?"

"Yes," they confirmed, "he wants to see something live in time for the upcoming Brazilian election in a few months!"

"Do you need *all* of this to go live for the election?" I asked.

As soon as I asked the question, I could see light bulbs click on. Of course they didn’t need *everything*. Up 'til now, they’d been focused on identifying sequence and dependency assuming they’d need to build everything. They did, but it was really a question of when. They shifted their thinking to focus on outcomes.

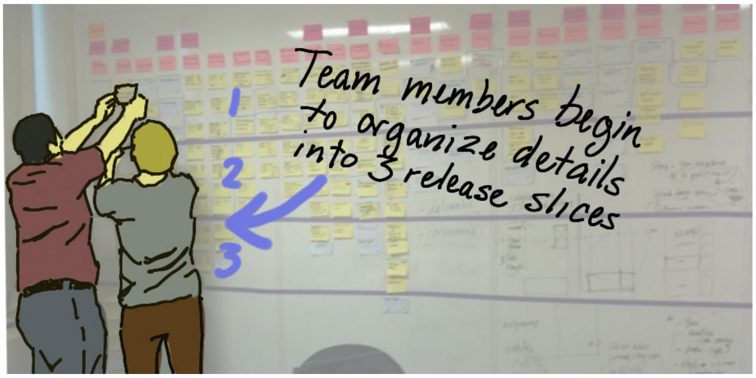
Focus on what you hope will happen outside the system to make decisions about what’s inside the system.

Globo.com focused on Brazilian elections. The teams thought specifically about the great *outcome* of successfully impressing visitors, advertisers, and Globo’s parent media company with newer, sexier styles of interactive content throughout the election. If they did that, they’d have a win.

This wasn’t their first time facing down an impossible delivery date. They thought about it for a bit, and realized that they definitely needed to go live with the news website, and potentially some other things that supported it because that’s where visitors and others would go to monitor Brazilian elections. Focusing on the news website meant paying attention to innovative ways to show real-time election data and breaking news stories faster. And, of course, there was a newer, up-to-date visual design overlaying everything.

Slice Out a Minimum Viable Product Release

The teams grabbed a roll of blue painter’s tape and stretched lines across the map left to right to make horizontal slices. They then went to work moving cards up and down, above and below the blue lines to designate which things needed to be done in the first slice, and which could be done later.


Team members begin to organize details into 3 release slices
1
2
3

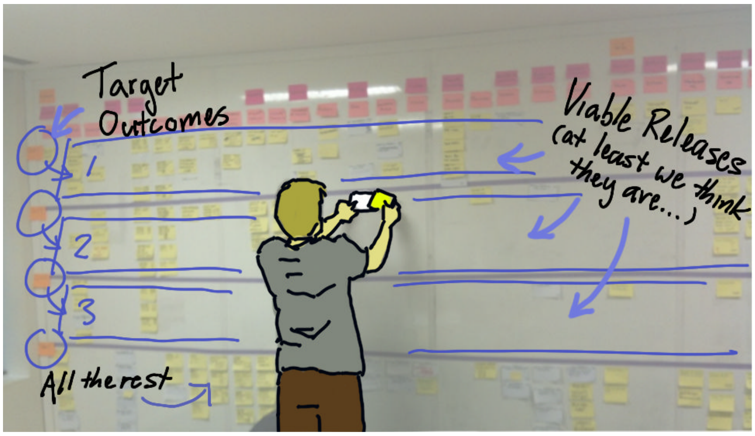
The thought process worked a bit like this: *If we go live for the Brazilian elections, a lot of people in Brazil will see this thing. It's going to land with a splash. It's going to affect these websites of ours and we'll look good. Everything in this slice is what users will need to be able to do after the software is released so they can make that splash.*

Focus on outcomes—what users need to do and see when the system comes out—and slice out releases that will get you those outcomes.

Slice Out a Release Roadmap

The map contained innovative things that would improve all Globo.com's web properties. But it really was going to take a long time to get all of it done. Hitting the market window that the elections created would be too big an opportunity to miss. Focusing on that helped Globo identify the first release.

The teams then went to work thinking about what kinds of web properties and market events should anchor the next releases. They posted sticky notes to the left of each slice with a few words describing their intention for each release slice—their target outcomes. Then they continued to move cards up and down into their correct slices.



In the end, they had an incremental release strategy that let them tackle all the work they needed to do to replace the whole content management system over time, and in such a way that they saw real benefit with each release. If they read down the left edge of the map, they had a list of named releases, each with specific target outcomes. This is a *release roadmap*.

Notice how the list isn't a bunch of features. It's a list of real-world benefits—because, remember, your job isn't to build software, it's to change the world. The hard part is choosing which people you want to change the world for, and how.

Focusing on specific target outcomes is the secret to prioritizing development work.

And the opposite is true as well; that is, if you don't know what your target outcomes are—the specific benefits you're trying to get—then prioritization is close to impossible.

Don't Prioritize Features—Prioritize Outcomes

Notice also how the Globo teams started with a big goal of replacing their entire content management system. Replacing the content management system is the output, the stuff they were going to deliver. Doing that would result in lots of positive outcomes. The secret to breaking down that really big chunk of output was to focus on a smaller, specific outcome.

Remember: behind outcomes are specific behavior changes for specific people engaged in specific activities. By focusing on the upcoming Brazilian elections, Globo chose to focus on the people who follow the news, especially those looking for up-to-the-minute election details. But, in placing its focus on those people, it left out soap opera lovers, sports lovers, and lots of other types of users. Those other people would have to be satisfied with the current site for a while longer. Remember, you can't please everyone all the time.

This Is Magic—Really, It Is

I may be easily impressed, but slicing is one of the coolest things about organizing software ideas into a story map.

Many times I, and the teams I've worked with, have placed all our ideas about the perfect product into a map and been overwhelmed by the amount of work we'd have if we created it all. It *all* seems important. But then we step back and think about the specific people who will use our product, and what they'll need to accomplish to be successful. We distill that into a sentence or two. Then we carve away everything we don't need, and we're shocked at how small our viable solution really is. It's *magic*.

Gary from Chapter 1 went on to do something similar with his product. He eventually narrowed focus to the band manager, the fan, and Mimi's internal administrator—because you've got to keep the site running. Gary chose to leave out venue managers and production musicians. In the end, what he realized is that by focusing on just those few people and the activity of promoting, he ended up with a great email promotion platform. So, for those of you who are Mimi users today, that's the experience you know.

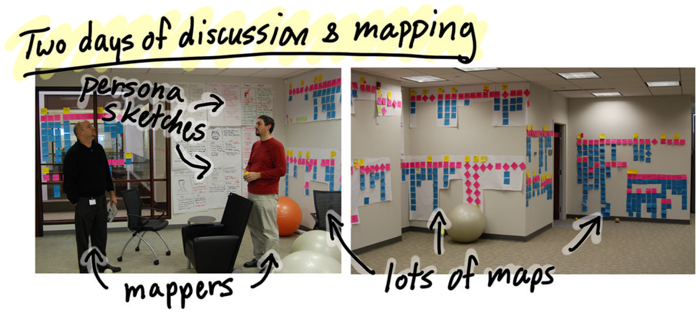
Externalizing our thinking in a big visible map makes all these steps easier. It makes it possible for lots of people to collaborate to accomplish it.

Finding a Smaller Viable Release

Chris Shinkle, SEP

FORUM Credit Union is one of the nation's largest and most technologically progressive credit unions. Although the company had a competent and creative development culture, it approached SEP to build a new online banking system that rivaled existing commercial off-the-shelf (COTS) solutions. Its goals included adding capabilities for its mobile banking, personal finance management, and text banking.

SEP kicked off the engagement with a two-day, collaborative story-mapping discovery session that included outcomes, personas, and story mapping. The session helped facilitate a structured conversation to prioritize a large set of feature ideas. The outcomes and personas were not enough, however, to prioritize the stories. By the end of two days, the story map covered nearly two walls in the 1,000+ square-foot development area!



After the story map was constructed, SEP guided the FORUM stakeholders through a simple prioritization model:

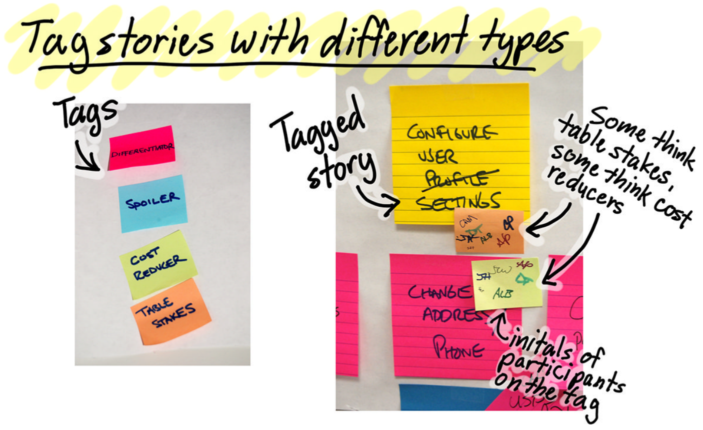
Differentiator
: A feature that set them apart from their competition

Spoiler
: A feature that is moving in on someone else's differentiator

Cost reducer
: A feature that reduces the organization costs

Table stakes
: A feature necessary to compete in the marketplace

SEP indicated each story's category using different color sticky notes. Interesting discussions emerged—in fact, some stakeholders' differentiators were another stakeholder's table stakes. It was clear many of these conversations were happening for the first time! The story map with the prioritization model enabled conversations to happen that had not happened before. It helped guide the team to come to a shared understanding around priorities.


Tag stories with different types
Tags
DIFFERENTIATOR
SPOILER
COST REDUCER
TABLE STAKES
Tagged story
CONFIGURE USER ~~PROFILE~~ SETTINGS
CHANGE ADDRESS PHONE
Some think table stakes, some think cost reducers
initials of participants on the tag

After labeling the stories, SEP used a voting system to help stakeholders converge the ideation and discussion into the most meaningful set of outcome-focused features. To everyone's surprise, several stories were deemed postpone-worthy or unnecessary. Rough calculations revealed several hundred thousand dollars in savings before one line of code was written.

When asked about using story mapping to kick off the project, Doug True, FORUM's CEO, said, "When we first kick-started this project with a story mapping process including the use of personas, I was skeptical. Specifically, I was concerned with the time invested to this softer side of the project. On the second day it clicked and the worthiness of the time materialized. In fact, now, I couldn't imagine pursuing a project of this scope and member-facing impact without such a process."

Why We Argue So Much About MVP

There's a term that's been kicking around in the software development industry for a long time: *minimum viable product*, or simply MVP.

Frank Robinson is credited with originally coining the term *MVP*, but these days definitions from Eric Ries and Steve Blank dominate. In spite of multiple smart people trying to define the term, everyone still seems confused—including me. Every organization I run into that uses that term means something slightly different. Even people in the same organization and the same conversation often mean different things.

Like most words in the dictionary, it has multiple meanings. I'm going to give you three definitions for the term: a bad one, and two good ones.

Here's the bad one:

Minimum viable product is not the crappiest product you could possibly release.

And the MVP *isn't* the product that your users could use, but only in the simplest of circumstances, and only if they had a high threshold for pain. I commonly see organizations rationalizing their bad product decisions with the argument that someone could use the product, when it's clear to everyone involved that they probably wouldn't choose to.

If Globo.com had used this definition for slicing out its first release, it would have had a negative outcome. No one would have been impressed. The brand would have been hurt, and the company would have been worse off than if it had released nothing at all.

When we use the term *viable* when talking about a living organism, it means that an organism can survive in the world on its own without dying. And when we talk about software, we mean the same thing.

The minimum viable product is the smallest product release that successfully achieves its desired outcomes.

I like this definition best. *Minimum* is a subjective term. So be specific about who it's subjective to—because it's not you. Be specific about who your customers and users are, and what they need to accomplish. What's *minimum* to them? I promise you, it'll help the conversation hugely. It's still a tough conversation. But the alternative is the "HiPPO" method—the "highest paid person's opinion." That one sucks worse.

The term I prefer these days is *minimum viable solution*. Most of the things I work with organizations on aren't whole, new products. They're new features or capabilities, or improvements on features already out there. So the term *solution* seems to make more sense. So let me revise my definition:

> *The minimum viable solution is the smallest solution release that successfully achieves its desired outcomes.*

Here comes the hard part…

We're just guessing.

When we slice out a bunch of software functionality and call it a minimum viable solution, we don't really know if it is.

The problem with outcomes is that you can't really observe them until things come out. When you slice out a release, you're forced to hypothesize what will happen. You might have to guess about what customers will buy your product, what users will choose to use it, if they can use it, and what's feasible to build in the time you have. You're forced to guess at how much will make them happy. That's a lot of guessing.

This sucks, because if you guess too low, well, that's less than minimal, and you've failed. If you guess too high, which many people do to hedge their bets, then you've spent too much money and often risk not getting things done at all. And worst of all, you could be just dead wrong, and no amount of what you ship will matter at all.

It's no wonder that the "crappiest product you could possibly release" definition still thrives. Because that's the one we don't need to guess about.

The New MVP Isn't a Product at All!

I know that some of you may have been feeling progressively twitchy over the last couple of chapters. You may have thought to yourself, *Jeff's overlooking the most important thing of all!* And you might be right. Some of the most important things you can discuss during story and story map conversations are:

- What are our biggest, riskiest assumptions? Where is the uncertainty?
- What could I do to learn something that would replace risks or assumptions with real information?

This leads me to my third definition of MVP, as popularized by Eric Ries in his book *The Lean Startup* (Crown Business). Eric learned the hard way, as most of us do, that we're just guessing. Eric worked for a company that released a product that it thought was viable, but it was wrong. Intelligently, he changed his strategy to focus on learning—to focus on validating all those assumptions the company had made in its first MVP release. Eric makes the important point that we need to create smaller experiments, prototypes that test our hypothesis about what's minimal and viable. And if you adopt Eric's way of thinking, which you should, your first product should really be an experiment —and the one after that, and the one after that, until you really prove that you've got the right product.

A minimal viable product is also the smallest thing you could create or do to prove or disprove an assumption.

While it was pretty cool that the folks at Globo.com were able to create a plan to build less, they weren't fooling themselves. They knew there was lots left to learn to prove their assumptions were good. From here they and everyone else need to create a plan to learn more. And that's where we'll pick up our story in the next chapter.

CHAPTER 3

Plan to Learn Faster

This is my friend Eric, standing in front of his backlog and task board in his team room. He's a product owner working hard with his team to build a successful product, but right now it's not. That doesn't worry Eric, though. He has a strategy for making his product successful. And so far it's working.

Eric works for a company called Liquidnet. Liquidnet is a global trading network for institutional investors. Long before Eric came to stand in front of the board in the picture, someone at his company identified a group of customers Liquidnet could serve better, along with a few ideas of how to do that. Eric is part of a team that took those ideas and ran with them. That's what product owners do. If you thought they were always acting on their own great ideas, well, you're wrong. One of the hard parts of being a product owner is taking ownership of someone else's idea and helping to make it successful, or proving that it isn't likely to be. The best product owners, like Eric, help their entire team take ownership of the product.

Start by Discussing Your Opportunity

Eric didn't start his work by building a backlog of user stories. He started with the big idea someone had, and treated it like an opportunity for his company—because it was. He had conversations with leadership in his company to understand more. They discussed:

- *What* is the big idea?
- *Who are the customers?* Who are the companies we think would buy the product?
- *Who are the users?* Who are the types of people inside those companies we think would use the product, and what would they be using it for?
- *Why would they want it?* What problems would it solve for customers and users that they couldn't solve today? What benefit would they get from buying and using it?
- *Why are we building it?* If we build this product and it's successful, how does that help us?

Eric needed to build shared understanding with others in his organization before he could take ownership of the opportunity. He knew he was going to need to tell the story of this product many times over the coming months, so he'd better get the big stuff right now.

Your first story discussion is for framing the opportunity.

Validate the Problem

Eric trusts his leadership's intuition, but he knows that this big idea is a hypothesis. He knows the only way to be sure the idea will succeed is when they actually see it succeed.

He first spent time talking to customers and users directly to really learn about them. Along the way he validated that there really were customers who had the problem, and they really were interested in buying a solution. Eric talked to the people who'd likely use the product. They didn't have the product today, and had only poor workarounds to address the problems the new product idea would solve.

Validate that the problems you're solving really exist.

While Eric's been talking with customers and users, he's been building up a pool of people he thinks are good candidates to try his new software. Some companies refer to these people as *customer development partners*. Keep track of this detail, because it's going to come up later in the story.

Actually, during this stage, it wasn't just Eric. Eric was working with a small team of others who spent lots of time talking to their customers, and, in doing so, found that solving the problem wasn't so easy—and that there were other problems that needed to be solved first. The important thing for you to take away is that the more they learned, the more the original opportunity was changed—eventually, a lot. It's lucky they didn't just get to work building what they were told to. That wouldn't have served their customers or their organization.

By now Eric and his team, after talking to customers, had specific ideas for the type of solution they could build that users could use, and by doing so get the benefit their employers wanted. Now, here's where Eric and his team could have gone "all in"—where they could have bet it all. They could have built a backlog of stories that described their solution and set a team to work building it. Because they're smart people, they'd have used a story map to move from the big idea to the specific parts to build. But, because they're *really* smart, the last thing they're going to do at this point is to build software.

Prototype to Learn

It's around here that Eric began to act as the owner for this product. He moved to envision his solution first as a bunch of simple narrative stories—*user scenarios*. Then he moved to envisioning the idea as a simple wireframe sketch. And then he created a higher-fidelity prototype. This wasn't working software. It was a simple electronic prototype, created with a simple tool like Axure, or maybe even PowerPoint.

All of these are learning steps for Eric. They help him envision his solution. Ultimately, he wants to put his solution in front of his users to see what they think. But he knows he first needs to feel confident it solves their problems before he puts it in front of them.

Sketch and prototype so you can envision your solution.

Now, I've hidden an important detail from you. Eric was actually an interaction designer. He's the kind of designer who's used to spending time with customers and users, and used to building these simple prototypes. But, for this new product, he's also the product owner—the one ultimately responsible for the product's success. There are other product owners in Eric's company who don't have his design skills, and they very sensibly pair with designers to help with both interviewing users and envisioning solutions.

Eric did eventually bring prototypes back to users. And I wasn't there, so I don't know what really happened for Eric. But I've been in these situation lots of times, and I'm always surprised about what I learn from the people who'll really use my solution. All I can tell you is, be prepared for surprises and bad news. In fact, celebrate the bad news, because you could have received the same bad news months later, after you'd built the software. That's when it really sucks. Right now, it's cheap to make changes, and you should. And Eric did.

Prototype and test with users to learn whether your solution is valuable and usable.

After iterating his solution many times and showing it to his customers, Eric was confident he had a pretty good solution idea. Surely now he could get that backlog built, and get his team of developers to work turning that prototyped solution into real working software. But Eric's not going to do that. Well, not exactly that. That's a bigger bet than he's willing to take.

Watch Out for What People Say They Want

Eric has prototyped what he believes is a viable solution. But he's not really sure if it's minimal—because he showed people lots of cool ideas. And if you show people all the cool ideas, of course they'll love them. But Eric knows his job is to minimize the amount he builds and still keep people happy. How much could he take away and still have a viable solution?

Eric also knows something else that's a bit disturbing. He knows the people who said they'd like it and use it are just guessing, too.

Think back to when you've bought something yourself. You may have looked at the product. You might have watched a salesperson demonstrate the cool features. You may have tried out the cool features for yourself, and you could imagine really using and loving the product. But when you bought the product and actually started using it, you found the cool features didn't matter so much. What really mattered were features you hadn't thought about. And, worst of all, maybe you didn't really need the product that much after all. OK, maybe it's just me I'm talking about. But I've got lots of stuff in my garage that I wish I'd never bought.

Back to Eric. He knows his customers and users can imagine the product would be great to use, and knowing that gives him the conviction to up his bet. But the real proof is when those people actually *choose* to use it every day. That's the real outcome he's looking for—and the only outcome that'll get his company the benefit it really wants. And it's going to take more than a prototype to learn that.

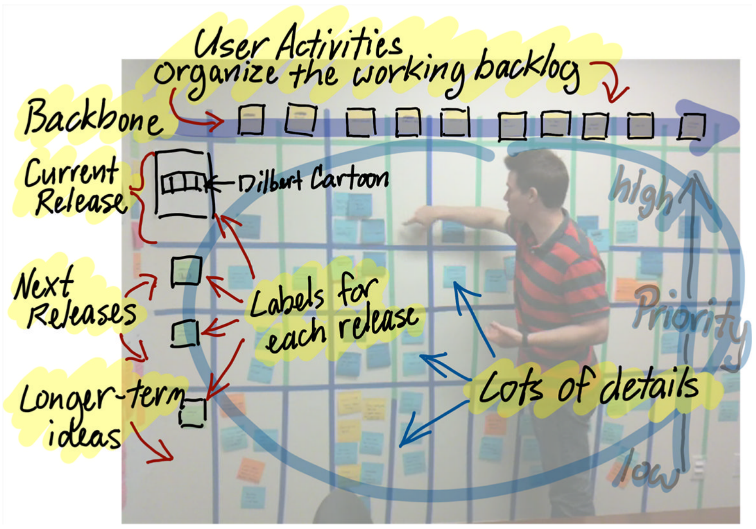
Build to Learn

Now, here's where Eric gets to show how smart he really is.

Eric and his team actually do get to work building software. But their first goal isn't to build a minimum viable product. Actually, it's to build something less than minimal—just enough that potential users could do something useful with it. This is a product that wouldn't impress too many people, and they might even hate it. It's definitely not a product you'd want your marketing and sales people out there pitching. In fact, the only people you'd want to see this product are people who may one day use the product, and honestly care about finding a product that solves their problem.

It just so happens that Eric has a small group of people just like that. It's the customers and users he worked with earlier when he was learning about and validating the problem. They're his development partners. They're the ones who gave feedback on early prototypes. And there's a subset of them that Eric believes can best help him learn. They're the ones he'll put this first, less-than-minimum—and definitely not viable—product in front of. He hopes they'll become his early adopters.

And that's what he did.



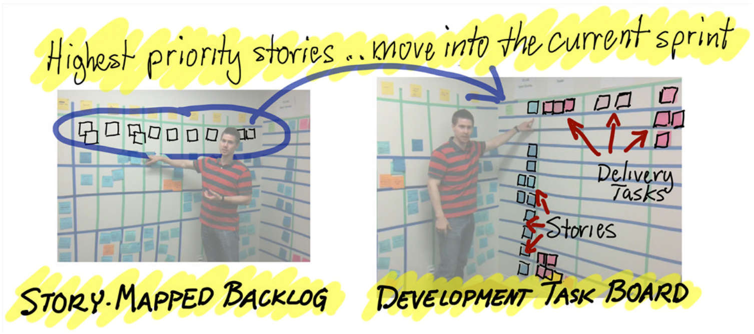
This is Eric pointing out a slice of his current backlog. When this picture was taken, he'd already released software to his development partners. After he did, he made a point of talking to them to get their feedback. His team also built in some simple metrics so they could measure whether people were really using the software, and what they did in the software specifically.

Eric knows that people are polite. They may say they like a product, but then never use it. The "using it" is the real outcome he wants, and polite isn't helping him. Eric also knows some people are demanding.

They may list all the problems the product has, or complain about bugs, but the metrics may be telling us that they use it every day anyway. And that's a good thing, in spite of all their complaining. The complaining is good, too, because it gives Eric ideas about where his next improvements should be.

Eric's backlog is organized as a story map with the backbone in yellow stickies across the top. Those yellow stickies have short verb phrases on them that tell the big story of what his users will do in the product, but at a high level. Below it are all the details—the specific things they'll do and need to really use the product. While the details he and his team work on change from release to release, the backbone stays pretty consistent.

The top slice, above the tapeline, is the one Eric and his team are working on right now. This release will take Eric two sprints. He's using a Scrum development process where his sprints are two-week timeboxes. So two sprints equate to basically a month. Below that are slices running down the board. The next slice contains what they think the next release might be, and so on. To the left of each slice, just as with the Globo.com team, hangs a sticky note with the release name and a few words about what they want to learn in this release. Except for the top release, which has a *Dilbert* cartoon posted over it. It's an inside joke in their team that I wasn't in on.



If you look closely, the top of that current slice is sort of cleaned out. I've drawn in some sticky notes where they used to be. But they're not there anymore because those things that were on the top are the first things his team will build. As the team members worked together to plan their work, they removed those sticky notes and placed them on a task board to the right of the story-mapped backlog. That task board shows the stories they're working on now in this sprint, along with delivery task—the specific things that the developers and testers will need to do to turn the ideas in the story into working software.

One finer point of Eric's story-mapped backlog, and one that proves he's smart, is the thickness of that topmost slice. It's twice as thick as the slices below it. When Eric and his team finish a slice and deliver it to their development partners—what they call their *beta customers* —they'll move the sticky notes up from the slice below. When they do, they'll have lots more detailed discussion about this next sliced-out release. They'll play What About to find problems and fill in details. They'll talk about some of the ideas in the next release, and this discussion may result in their splitting the big idea into two or three smaller ideas. And then, they'll need the vertical height in that slice to prioritize—to make choices about what to build first.

See how smart they are?

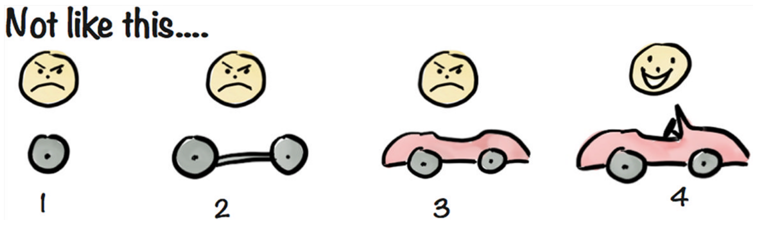
Iterate Until Viable

Eric may have started this whole process with an idea about what the minimum viable product might be, but he's purposely built something less than minimal to start with. He's then adding a bit more every month. He's getting feedback from his development partners—both the subjective stuff from talking to them, and the more objective stuff he gets from looking at data.

He'll keep up this strategy, slowly growing and improving the product, until his development partners actually start using the product routinely. In fact, what Eric's hoping for is that they become people who'd recommend the product to other people—real reference customers. When they do, that's when he knows he's found minimum *and* viable. And that's when the product is safe to market and sell like crazy. If Eric and his team had tried to sell it before, they'd have ended up with lots of disappointed customers—people a lot less friendly than those with whom he built personal relationships throughout this process.

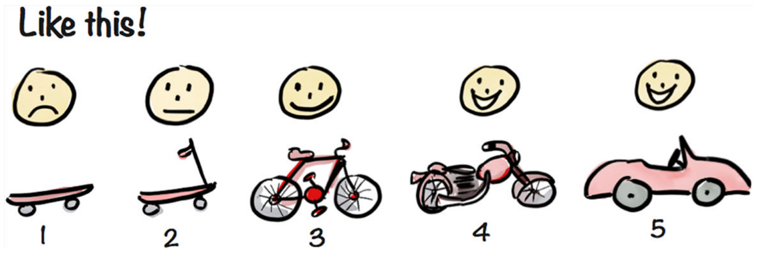
How to Do It the Wrong Way

What Eric could have done is to take his last, best prototype, break it down into all its constituent parts, and start building it part by part. Many months later, he'd have had something to release. And he'd have learned then if his big guess was right. You'll need to trust me on this, but it wouldn't have been—because it rarely is.



This is a simple visualization made by my friend Henrik Kniberg. It beautifully illustrates a broken release strategy where at every release I get something I can't use, until the last release when I get something I can.

Henrik suggests this alternative strategy:



If I plan my releases this way, in each release I deliver something people can actually use. Now, in this silly transportation example, if my goal is to travel a long distance and carry some stuff with me, and you gave me a skateboard, I might feel a bit frustrated. I'd let you know how difficult it was to travel long distances with that thing—although it was fun to goof around with it in the driveway. If your goal was to leave me delighted, you might feel bad about that. But your real goal was to learn, which you did. So that's good. You learned I wanted to travel farther, and if you picked up on it, you also learned I valued having fun.

In Henrik's progression, things start picking up at around the bicycle release because I can actually use it as adequate transportation. And, at about motorcycle level, I can *really* see this working for me—and I'm having fun too. That could be minimum and viable for me. If I really love the motorcycle thing, maybe my next best step would be a bigger, faster Harley-Davidson, and not a sports car. I'm headed for a midlife crisis right now and that Harley is sounding pretty good. But it's after I try the motorcycle, and we both learn something from that, that we can best make that decision.

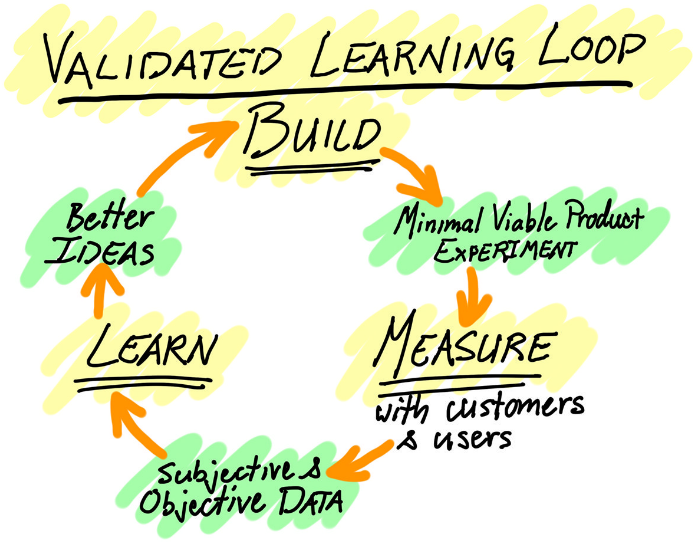
Treat every release as an experiment and be mindful of what you want to learn.

But what about other folks who need to travel longer distance, and who have kids? For *that* target market, none of these would be good choices.

Always keep your target customers, users, and the outcomes you're hoping for in mind. It's really tough to get the same great outcome from all types of users. So focus.

Validated Learning

What my friend Eric did is apply a *validated learning strategy*—one of the important concepts in Lean Startup thinking. Eric knew that the problems he was solving, the customers and users he was solving them for, and the solutions he had in mind were all assumptions. Lots of them were pretty good assumptions. But they were assumptions just the same. Eric set out to understand the assumptions and then validate them, moving from the problems customers and users faced to the solutions he had for them. At each step he did or built something with the explicit goal of learning something.



What Eric did is the heart of the build-measure-learn loop described by Eric Ries. And by Ries's definition, each release that Eric shipped was a minimum viable product. But you can see that it wasn't viable in the eyes of his target customers and users—at least, not yet. For that reason, I like referring to Ries's MVP as a *minimum viable product experiment*—or MVPe for short. It's the smallest thing I could build to learn something. And what I learn is driving toward understanding what's really viable in the eyes of my target customers and users.

Eric used lots of tools and techniques along the way. But telling stories using words and pictures was always part of the way he worked. Using a map to organize his stories helped him keep his customers, users, and their journey in mind as he iteratively improved his product to viable.

I like using the term *product discovery* to describe what we're really doing at this stage. Our goal isn't to get something built; rather, it is to learn if we're building the right thing. It just so happens that building something to put in front of customers is one of the best ways to learn if we're building the right thing. I borrow my definition of *discovery* from Marty Cagan.[1] And my definition of *discovery* includes Lean

1. Marty first described what he means by *product discovery* in this 2007 essay. He later describes it in more detail in his book *Inspired: How to Create Products Customers Love* (SVPG Press).

Startup practice, Lean User Experience practice, Design Thinking practice, and loads of other ideas. And what I do during discovery continues to evolve. But the goal stays the same: to learn as fast as possible whether I'm building the right thing.

Really Minimize Your Experiments

If we recognize that our goal is to learn, then we can minimize what we build and focus on building only what we need to learn. If you're doing this well, it means that what you build early may not be production ready. In fact, if it is, you've likely done too much.

Here's an example: when I was a product owner for a company that built software for large, chain retailers, I knew my products needed to run on a big Oracle database on the backend. But the database guys were sometimes a pain for me to work with. They wanted to scrutinize every change I made. Sometimes simple changes would take a week or more. That slowed down my team and me too much. The database guys' concerns made sense, since all the other applications depended on that database. Breaking it was really risky for everyone. But they had a well-oiled process for evaluating and making database changes —it just took a long time.

The riskiest part for me was making sure my product was right. So we built early versions of software using simple, in-memory databases. Of course, they wouldn't scale, and we could never release our early versions to a large general audience. But our early minimum viable product experiments (we didn't call them that then) allowed us to test ideas with a small subset of customers and still use real data. After several iterations with customers, and after we found a solution we believed would work, we'd then make the database changes and switch our application off the in-memory database. The database guys liked us too, because they knew that when we made changes, we were confident they were the right ones.

Let's Recap

Gary used a map to get out of the flat-backlog trap and see the big picture of his product, and then to really start focusing on who it was for and what it should be.

The teams at Globo.com used a map to coordinate a big plan across multiple teams and slice out a subset of work they believed would be a viable solution.

Eric used a map to slice out less-than-viable releases into minimum viable product experiments that allowed him to iteratively find what would be viable.

There's one last challenge that seems to plague software development, and that's finishing on time. Suppose you're confident that you have something that should be built. And suppose others are depending on it going live on a specific date. There's a secret to finishing on time that's been known by artists for centuries. In the next chapter, we'll learn how to apply it to software.