

Principles of Shaping

When we shape the work, we need to do it at the right level of abstraction: not too vague and not too concrete. Product managers often err on one of these two extremes.

Wireframes are too concrete

When design leaders go straight to wireframes or high-fidelity mockups, they define too much detail too early. This leaves designers no room for creativity. One friend put it this way:

I'll give a wireframe to my designer, and then I'm saying to her: "I know you're looking at this, but that's not what I want you to design. I want you to re-think it!" It's hard to do that when you're giving them this concrete thing.

Over-specifying the design also leads to estimation errors. Counter-intuitive as it may seem, the more specific the work is, the harder it can be to estimate. That's because making the interface *just so* can

require solving hidden complexities and implementation details that weren't visible in the mockup. When the scope isn't variable, the team can't reconsider a design decision that is turning out to cost more than it's worth.

Words are too abstract

On the other end of the spectrum, projects that are too vague don't work either. When a project is defined in a few words, nobody knows what it means. "Build a calendar view" or "add group notifications" sound sensible, but what exactly do they entail? Team members don't have enough information to make trade-offs. They don't know what to include or leave out. A programmer who worked in that situation said:

You're solving a problem with no context. You have to be a mind reader. It's like: "we'll know it when we see it."

Concerning estimation, under-specified projects naturally grow out of control because there's no boundary to define what's out of scope.

Case study: The Dot Grid Calendar

Let's look at an example of how to shape a project at the right level of detail.

We launched version three of Basecamp without a calendar feature. It had a "schedule" feature that just listed events one after the other without any kind of monthly, weekly or daily grid.

Soon after launch, customers started asking us to "add a calendar" to Basecamp. We had built calendars before and we knew how complex they are. It can easily take six months or more to build a proper calendar.

These are the kinds of things that make a calendar complicated:

- Dragging and dropping events between cells to move them
- Wrapping multi-day events around the edge of the screen
- Different views for monthly, weekly, or daily time scales
- Dragging the edge of an event to change its duration
- Color coding events for different categories
- Handling different expectations for desktop vs. mobile interactions

Past versions of Basecamp had calendars, and only about 10% of customers used them. That's why we didn't have the appetite for spending six months on a calendar. On the other hand, if we could do something to satisfy those customers who were writing us in one six week cycle, we were open to doing that.

With only six weeks to work with, we could only build about a tenth of what people think of when they say "calendar." The question became: which tenth?

We did some research (discussed in the next chapter) and narrowed down a use case that we wanted to solve. We eventually arrived at a promising concept inspired by calendars on phones. We could build a two-month, read-only grid view. Any day with an event would have a dot for each event. A list of events would appear below the calendar, and clicking a day with a dot would scroll the events for that day into view. We called it the Dot Grid.

The Dot Grid wasn't a full-featured calendar. We weren't going to allow dragging events between days. We weren't going to span

multi-day events across the grid; we'd just repeat the dots. There'd be no color coding or categories for events. We were comfortable with all these trade-offs because of our understanding of the use case.

This is the level of fidelity we used to define the solution:

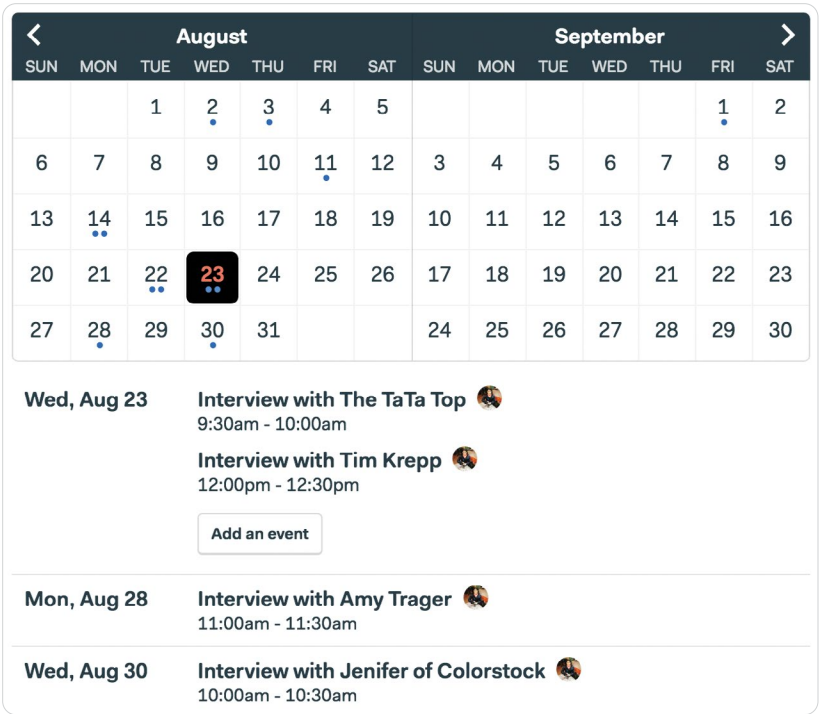


Rough sketch of the Dot Grid concept

Note how rough the sketch is and how many details are left out. The designer had a lot of room to interpret how this should look and feel.

At the same time, note how specific the idea is. It’s very clear how it works, what needs to be built, what’s in and what’s out.

At the end of the project, the finished work that the designers and programmers created looked like this:



Screenshot of the Dot Grid when it launched

This small example highlights a few properties of shaped work.

Property 1: It’s rough

Work in the shaping stage is rough. Everyone can tell by looking at it that it’s unfinished. They can see the open spaces where their contributions will go. Work that’s too fine, too early commits

everyone to the wrong details. Designers and programmers need room to apply their own judgement and expertise when they roll up their sleeves and discover all the real trade-offs that emerge.

Property 2: It's solved

Despite being rough and unfinished, shaped work has been thought through. All the main elements of the solution are there at the macro level and they connect together. The work isn't specified down to individual tasks, but the overall solution is spelled out. While surprises might still happen and icebergs could still emerge, there is clear direction showing what to do. Any open questions or rabbit holes we could see up front have been removed to reduce the project's risk.

Property 3: It's bounded

Lastly, shaped work indicates what *not* to do. It tells the team where to stop. There's a specific appetite—the amount of time the team is allowed to spend on the project. Completing the project within that fixed amount of time requires limiting the scope and leaving specific things out.

Taken together, the roughness leaves room for the team to resolve all the details, while the solution and boundaries act like guard rails. They reduce risk and channel the team's efforts, making sure they don't build too much, wander around, or get stuck.

Who shapes

Shaping is creative and integrative. It requires combining interface ideas with technical possibilities with business priorities. To do that you'll need to either embody these skills as a generalist or collaborate with one or two other people.

Shaping is primarily design work. The shaped concept is an interaction design viewed from the user's perspective. It defines what the feature does, how it works, and where it fits into existing flows.

You don't need to be a programmer to shape, but you need to be technically literate. You should be able to judge what's possible, what's easy and what's hard. Knowledge about how the system works will help you see opportunities or obstacles for implementing your idea.

It's also strategic work. Setting the appetite and coming up with a solution requires you to be critical about the problem. What are we trying to solve? Why does it matter? What counts as success? Which customers are affected? What is the cost of doing this instead of something else?

Shaping is a closed-door, creative process. You might be alone sketching on paper or in front of a whiteboard with a close collaborator. There'll be rough diagrams in front of you that nobody outside the room would be able to interpret. When working with a collaborator, you move fast, speak frankly and jump from one promising position to another. It's that kind of private, rough, early work.

Two tracks

You can't really schedule shaping work because, by its very nature, unshaped work is risky and unknown. For that reason we have two separate tracks: one for shaping, one for building. During any six week cycle, the teams are building work that's been previously shaped and the shapers are working on what the teams might potentially build in a future cycle. Work on the shaping track is kept private and not shared with the wider team until the commitment

has been made to bet on it. That gives the shapers the option to put work-in-progress on the shelf or drop it when it's not working out.

Steps to shaping

Shaping has four main steps that we will cover in the next four chapters.

1. **Set boundaries.** First we figure out how much time the raw idea is worth and how to define the problem. This gives us the basic boundaries to shape into.
2. **Rough out the elements.** Then comes the creative work of sketching a solution. We do this at a higher level of abstraction than wireframes in order to move fast and explore a wide enough range of possibilities. The output of this step is an idea that solves the problem within the appetite but without all the fine details worked out.
3. **Address risks and rabbit holes.** Once we think we have a solution, we take a hard look at it to find holes or unanswered questions that could trip up the team. We amend the solution, cut things out of it, or specify details at certain tricky spots to prevent the team from getting stuck or wasting time.
4. **Write the pitch.** Once we think we've shaped it enough to potentially bet on, we package it with a formal write-up called a pitch. The pitch summarizes the problem, constraints, solution, rabbit holes, and limitations. The pitch goes to the betting table for consideration. If the project gets chosen, the pitch can be re-used at kick-off to explain the project to the team.



Set Boundaries

The first step of shaping is setting boundaries on what we're trying to do. The conversations we have are going to be entirely different if people think we're talking about a small improvement or a major redesign.

The conversation about building a feature always starts with a raw idea, like “customers are asking for group notifications.” Before we all go down the rabbit hole discussing ways we can solve it, we should first set some broad terms on the discussion to make it productive.

Setting the appetite

Sometimes an idea gets us excited right away. In that case we need to temper the excitement by checking whether this is really something we're going to be able to invest time in or not. If we don't stop to think about how valuable the idea is, we can all jump too quickly to either committing resources or having long discussions about potential solutions that go nowhere.

Other ideas are less exciting and feel more like a challenge we didn't ask for. The customer wants a calendar; we don't particularly want to build one, but we do feel we need to do something about the request.

Whether we're chomping at the bit or reluctant to dive in, it helps to explicitly define how much of our time and attention the subject deserves. Is this something worth a quick fix if we can manage? Is it a big idea worth an entire cycle? Would we redesign what we already have to accommodate it? Will we only consider it if we can implement it as a minor tweak?

We call this the appetite. You can think of the appetite as a time budget for a standard team size. We usually set the appetite in two sizes:

- **Small Batch:** This is a project that a team of one designer and one or two programmers can build in one or two weeks. We batch these together into a six week cycle (more on that later).
- **Big Batch:** This project takes the same-size team a full six-weeks.

In rare cases where the scope is so big that a six-week project isn't conceivable, we'll try to hammer it down by narrowing the problem definition. If we still can't shrink the scope, we'll break off a meaningful part of the project that we can shape to a six-week appetite.

Fixed time, variable scope

An appetite is completely different from an estimate. Estimates start with a design and end with a number. Appetites start with a number and end with a design. We use the appetite as a creative constraint on the design process.

This principle, called “fixed time, variable scope,” is key to successfully defining and shipping projects. Take this book for an example. It’s hard to ship a book when you can always add more, explain more, or improve what’s already there. When you have a deadline, all of a sudden you have to make decisions. With one week left, I can choose between fixing typos or adding a new section to a chapter. That’s the tension between time, quality, and scope. I don’t want to release a book with embarrassing typos, so I’ll choose to reduce the scope by leaving out the extra section. Without the pressure of the fixed deadline, I wouldn’t make the trade-off. If the scope wasn’t variable, I’d *have* to include the extra section. Then there’d be no time to fix the quality issues.

We apply this principle at each stage of the process, from shaping potential projects to building and shipping them. First, the appetite constrains what kind of a solution we design during the shaping process. Later, when we hand the work to a team, the fixed time box pushes them to make decisions about what is core to the project and what is peripheral or unnecessary.

“Good” is relative

There’s no absolute definition of “the best” solution. The best is relative to your constraints. Without a time limit, there’s always a better version. The ultimate meal might be a ten course dinner. But when you’re hungry and in a hurry, a hot dog is perfect.

The amount of time we set for our appetite is going to lead us to different solutions. We could model a whole set of database columns in the fancy version, or just provide a flat textarea in the simple version. We could redesign the main landing page to accommodate a new feature, or we could push it back to a screen with fewer

design constraints. We can only judge what is a “good” solution in the context of how much time we want to spend and how important it is.

Responding to raw ideas

Our default response to any idea that comes up should be: “Interesting. Maybe some day.” In other words, a very soft “no” that leaves all our options open. We don’t put it in a backlog. We give it space so we can learn whether it’s really important and what it might entail.

It’s too early to say “yes” or “no” on first contact. Even if we’re excited about it, we shouldn’t make a commitment that we don’t yet understand. We need to do work on the idea before it’s shaped enough to bet resources on. If we always say “yes” to incoming requests we’ll end up with a giant pile of work that only grows.

It’s important to keep a cool manner and a bit of a poker face. We don’t want to shut down an idea that we don’t understand. New information might come in tomorrow that makes us see it differently. On the other hand, showing too much enthusiasm right away can set expectations that this thing is going to happen. We may not be able to commit to it once we’ve put it into context with everything else we want to do.

Narrow down the problem

In addition to setting the appetite, we usually need to narrow down our understanding of the problem.

We once had a customer ask us for more complex permission rules. It could easily have taken six weeks to build the change she wanted. Instead of taking the request at face value, we dug deeper.

It turned out that someone had archived a file without knowing the file would disappear for everyone else using the system. Instead of creating a rule to prevent some people from archiving, we realized we could put a warning on the archive action itself that explains the impact. That's a one-day change instead of a six-week project.

Another example is the “calendar view” from the previous chapter. Everyone knows what a calendar view is. But unpacking it revealed tons of unknowns and decisions that would drastically affect the scope. If we only want to spend six weeks instead of six months building a huge calendar, how do we narrow it down?

In that case we flip from asking “What could we build?” to “What's really going wrong?” Sure, a calendar sounds nice. But what is driving the request? At what point specifically does someone's current workflow break down without this thing they're asking for?

Case study: Defining “calendar”

In the case of the calendar request, we called a customer who asked for this feature. Instead of asking her why she wants a calendar and what it should look like, we asked her *when* she wanted a calendar. What was she doing when the thought occurred to ask for it?

She told us she worked in an office with a big calendar drawn on a chalkboard wall. Her officemates marked when they were meeting clients in the handful of meeting rooms on the calendar. One day she was working from home. A client called and asked her to schedule a meeting. She had to drive to the office to look at the wall calendar. Traffic was terrible along the way, and in the end there wasn't a free space that worked for her client. She could have saved an hour in traffic and a lot of frustration if she had been able to check for open spots on the calendar from her computer at home.

The insight wasn't "computerize the calendar"—that's obvious. What we learned was that "see free spaces" was the important thing for this use case, not "do everything a calendar does."

This story, and others like it, gave us a specific baseline to design against. Basecamp had an agenda view of events. It worked for listing major deadlines and milestones but it wasn't good for resource scheduling because you couldn't see empty spaces on it. We narrowed down the need from "do everything a calendar does" to "help me see free spaces so I can figure out when to schedule something."

We didn't have a solution yet. But now we felt like we had a problem that was specific enough to spark an idea that could fit within our appetite. This led us to the simpler "Dot Grid" concept from the last chapter.

What if we can't figure out a specific pain point or use case? Our appetite can also tell us how much research is worthwhile. If it's not critical now and we can't get our hands around the problem, we'll walk away from it and work on something else. Maybe in the future a new request or story will pop up that gives us better insight into the problem.

Watch out for grab-bags

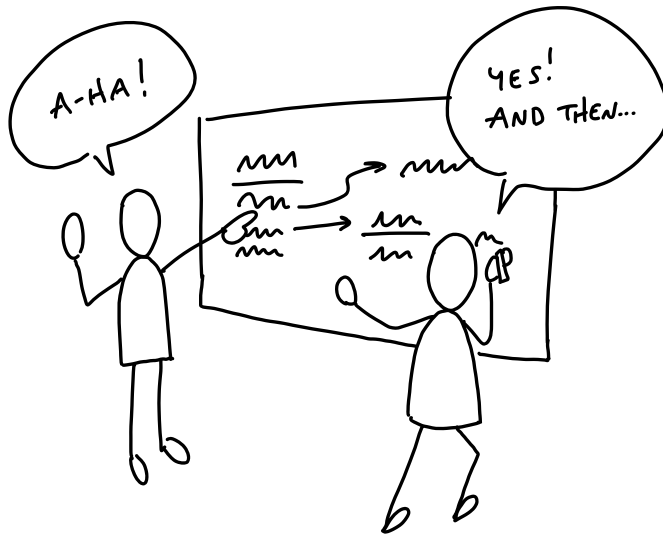
When it comes to unclear ideas, the worst offenders are "re-designs" or "refactorings" that aren't driven by a single problem or use case. When someone proposes something like "redesign the Files section," that's a grab-bag, not a project. It's going to be very hard to figure out what it means, where it starts, and where it ends. Here's a more productive starting point: "We need to rethink the Files section because sharing multiple files takes too many steps."

Now we can start asking: What’s not working? In what context are there too many steps? What parts of the existing design can stay the same and what parts need to change?

A tell-tale sign of a grab-bag is the “2.0” label. We made the mistake in the past of kicking off a “Files 2.0” project without really considering what that meant. Our excitement about improving a huge part of our app got the better of us. We know there were a lot of problems with our Files feature, but we didn’t ask ourselves what specifically we were going to do. The project turned out to be a mess because we didn’t know what “done” looked like. We recovered by splitting the project into smaller projects, like “Better file previews” and “Custom folder colors.” We set appetites and clear expectations on each project and shipped them successfully.

Boundaries in place

When we have all three things—a raw idea, an appetite, and a narrow problem definition—we’re ready to move to the next step and define the elements of a solution.



Find the Elements

Now that we have the constraints of an appetite and the problem we're solving, it's time to get from an idea in words to the elements of a software solution. There could be dozens of different ways to approach the solution for a problem. So it's important that we can move fast and cover a lot of different ideas without getting dragged down.

Move at the right speed

Two things enable us to move at the right speed at this stage.

First, we need to have the right people—or nobody—in the room. Either we're working alone or with a trusted partner who can keep pace with us. Someone we can speak with in shorthand, who has the same background knowledge, and who we can be frank with as we jump between ideas.

Second, we need to avoid the wrong level of detail in the drawings

and sketches. If we start with wireframes or specific visual layouts, we'll get stuck on unnecessary details and we won't be able to explore as broadly as we need to.

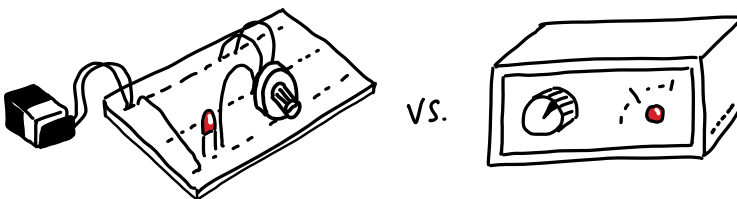
The challenge here is to be concrete enough to make progress on a specific solution without getting dragged down into fine details. The questions we're trying to answer are:

- Where in the current system does the new thing fit?
- How do you get to it?
- What are the key components or interactions?
- Where does it take you?

To stay on the right level of detail and capture our thoughts as they come, we work by hand using a couple of prototyping techniques: breadboarding and fat marker sketches. These allow us to quickly draw different versions of entire flows so we can debate the pros and cons of each approach and stay aligned with what we're talking about as we go.

Breadboarding

We borrow a concept from electrical engineering to help us design at the right level of abstraction. A breadboard is an electrical engineering prototype that has all the components and wiring of a real device but no industrial design.



Deciding to include an indicator light and a rotary knob is very different from debating the chassis material, whether the knob should go to the left of the light or the right, how sharp the corners should be, and so on.

Similarly, we can sketch and discuss the key components and connections of an interface idea without specifying a particular visual design. To do that, we can use a simple shorthand. There are three basic things we'll draw:

1. **Places:** These are things you can navigate to, like screens, dialogs, or menus that pop up.
2. **Affordances:** These are things the user can act on, like buttons and fields. We consider interface copy to be an affordance, too. Reading it is an act that gives the user information for subsequent actions.
3. **Connection lines:** These show how the affordances take the user from place to place.

We'll use words for everything instead of pictures. The important things are the components we're identifying and their connections. They allow us to play out an idea and judge if the sequence of actions serves the use case we're trying to solve.

Example

Suppose our product is an invoicing tool. We're considering adding a new "Autopay" feature to enable our customers' customers to pay future invoices automatically.

How do you turn Autopay on? What's involved? We can pick a starting point and say that the customer landed on an invoice. That's

our first place. We draw it by writing the name of the place and underlining it.

INVOICE

On the invoice, we're thinking we could add a new button to "Turn on Autopay." That's an affordance. Affordances go below the line to indicate they can be found at that place.

INVOICE
TURN ON
AUTOPAY

Where does that button go? Some place for setting up the Autopay. We don't have to specify whether it's a separate screen or a pop up modal or what. From a what's-connected-to-what standpoint (the topology) it's all the same. Let's draw a connection line from the button to the Setup Autopay screen.

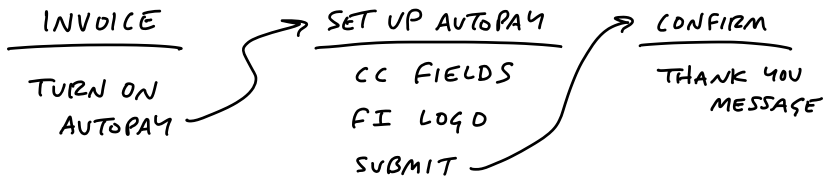
INVOICE
TURN ON
AUTOPAY → SET UP AUTOPAY

Now we can talk about what belongs on that screen. Do we ask for a credit card here? Is there a card on file already? What about ACH or other payment methods?

Just figuring out what to write under the bar starts to provoke

debates and discussions about what to build.

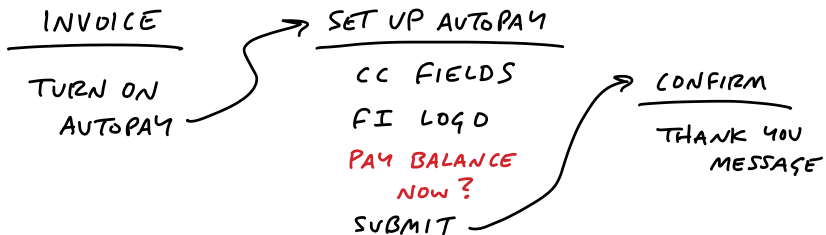
As we think it through, we decide we should ask for credit card details here and show the logo of the financial institution (an aspect of the domain in this specific product).



Straightforward enough. But wait — did we actually pay the original invoice or not? Hm. Now we have both functional and interface questions. What does enabling Autopay actually do? Does it apply only for the future or does paying with Autopay the first time also pay the current invoice? And where do we explain this behavior? We're starting to have deeper questions and discussions prompted by just a few words and arrows in the breadboard.

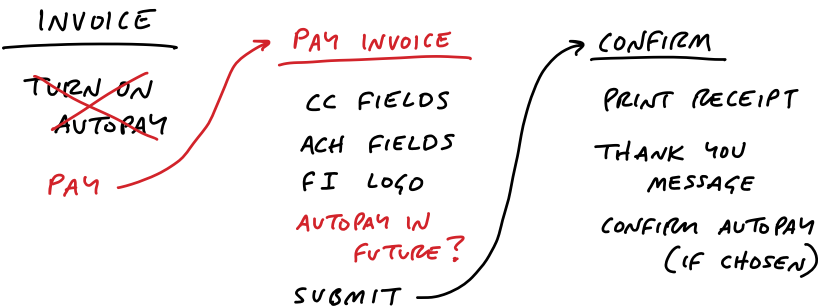
Since we're using such a lightweight notation, and we aren't bogged down with wireframes, we can quickly jump around and entertain different possibilities.

We could add an option to the Setup screen...



But now we’re complicating the responsibilities of the confirmation screen. We’re going to need to show a receipt if you pay your balance now. Should the confirmation have a condition to sometimes show a receipt of the amount just paid?

How about an entirely different approach. Instead of starting on an Invoice, we make Autopay an option when making a payment. This way there’s no ambiguity about whether the current amount is being paid. We could add an extra “Autopay was enabled” callout to the existing payment confirmation page.

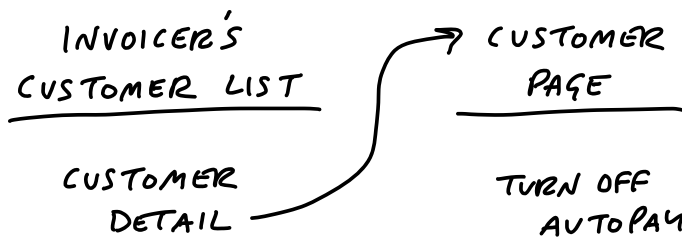


Sketching this out reminded us that the current payment form supports ACH in addition to credit card. We discuss and confirm that we can use ACH too.

What about after Autopay is enabled? How does the customer turn it off? Up to this point, many customers in the system didn’t have usernames or passwords. They followed tokenized links to pay the invoices one by one. One might naturally assume that now that the customer has something like Autopay, they need a username and password and some landing place to go manage it.

The team in this case decided that adding the username/password

flows was too much scope for their appetite at the time. Reflecting strategically on what they knew about their customers, they thought it would be quite alright if the invoicer's customers had to reach out to the invoicer and ask them to turn off the Autopay. In that case we could add a single option to disable Autopay in the customer detail page that we already offered to invoicers. We drew out the flow like this:



This example illustrates the level of thinking and the speed of movement to aim for during the breadboarding phase. Writing out the flows confronts us with questions we didn't originally think of and stimulates design ideas without distracting us with unimportant visual choices.

Once we get to a place where we play through the use case and the flow seems like a fit, we've got the elements we need to move on to start defining the project more clearly. We're getting more concrete while still leaving out a huge amount of detail.

Fat marker sketches

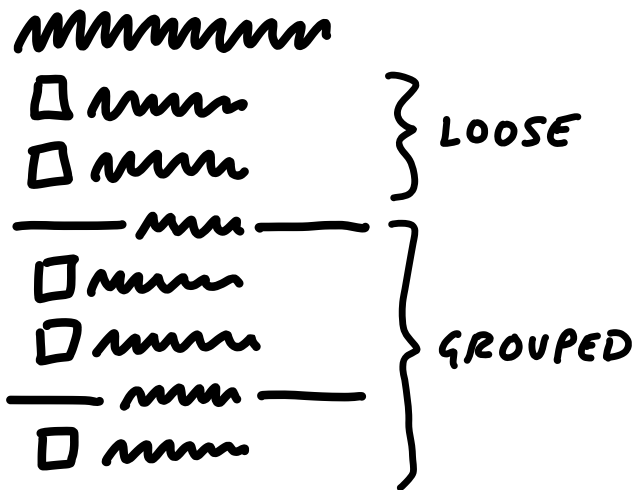
Sometimes the idea we have in mind is a visual one. Breadboarding would just miss the point because the 2D arrangement of elements is the fundamental problem. In that case, we still don't want to

waste time on wireframes or unnecessary fidelity. Instead we use fat marker sketches.

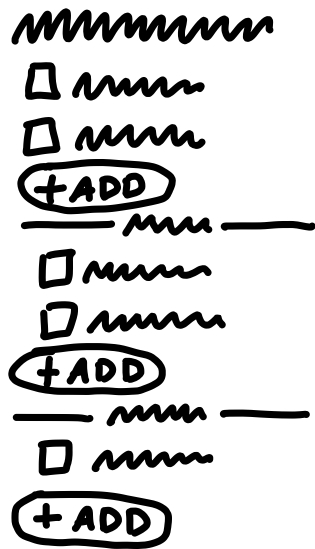
A fat marker sketch is a sketch made with such broad strokes that adding detail is difficult or impossible. We originally did this with larger tipped Sharpie markers on paper. Today we also do it on iPads with the pen size set to a large diameter.

Here’s an example. We found ourselves often creating fake to-dos in our Basecamp to-do lists that acted as dividers. We’d create an item like “--- Needs testing ---” and put items below it. We had the idea to make some kind of official divider feature in our to-do tool to turn the workaround into a first class function of to-do lists.

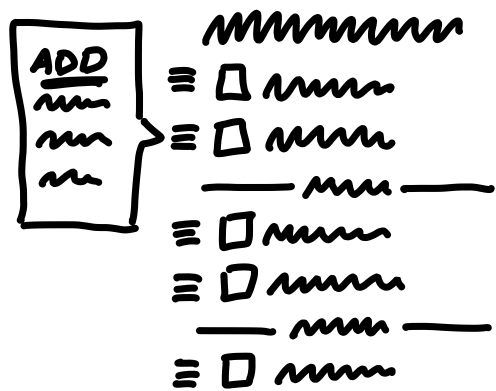
We had to work out what the implications of adding a divider were. We came up with a rough idea that adding a divider separates the list into “loose” to-dos above the divider and “grouped” to-dos below. Adding subsequent dividers adds more groups below the “loose” items at the top.



We could add items via some affordance within each group, including the “loose” group on top.



We were a little concerned the add buttons might break up the gestalt of the list, and the groups might all separate too much from the lists on the page. We talked about possibilities to place the “add” affordance inside of a menu that we already had to the left of each to-do item.



This notation is much less constraining than breadboards, which has downsides. We might sketch a sidebar and get attached to a layout element like that even though it's not a core element. But as long as we keep an eye on that we're still far better off than if we get sucked into the weeds by creating wireframes too early.

It may seem a little silly to call fat marker sketches a technique or a tool. The reason for calling them out is we too easily skip ahead to the wrong level of fidelity. Giving this rough early stage a name and using a specific tool for it helps us to segment our own creative process and make sure we aren't jumping ahead to detail a specific idea when we haven't surveyed the field enough.

Elements are the output

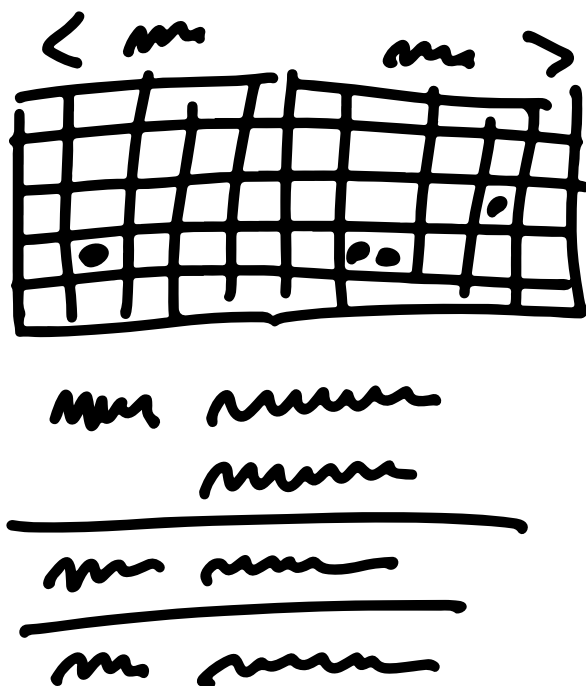
In the case of the Autopay example, we ended up with some clear elements:

- A new “use this to Autopay?” checkbox on the existing “Pay an invoice” screen
- A “disable Autopay” option on the invoicer's side

For the To-Do Groups project, the elements were:

- Loose to-dos above the first group belong directly to the parent
- Grouped to-dos appear below the loose to-dos
- We'd like to try an *add* affordance within each section, but if that doesn't work visually, we're ok with relying on the action menu for inserting to-dos into position.

Similarly, when we sketched the simplified solution for rendering events on a calendar grid, we used the fat marker approach.



This enabled us to work out the main elements of the solution:

- A 2-up monthly calendar grid
- Dots for events, no spanned pills
- Agenda-style list of events below that scrolls an event into view when you tap a dot

This list of elements is extremely narrow and specific compared to “monthly calendar.” Exactly the kind of narrowing we hope to accomplish through the shaping process.

Room for designers

Later, when it's time to involve a designer, you don't want to have to say "I know I drew it like this but ignore that...". Regardless of what you say, any specific mockups are going to bias what other people do after you—especially if you're in a higher position than them. They'll take every detail in the initial mockups as direction even though you didn't intend it.

Working at the right "level of abstraction" not only ensures we move at the right speed, it also leaves this important room for creativity in the later stages.

By leaving details out, the breadboard and fat marker methods give room to designers in subsequent phases of the project.

This is a theme of the shaping process. We're making the project more specific and concrete, but still leaving lots of space for decisions and choices to be made later. This isn't a spec. It's more like the boundaries and rules of a game. It could go in countless different ways once it's time to play.

Not deliverable yet

This step of shaping is still very much in your private sphere. It's normal for the artifacts at this point — on the wall or in your notebook — to be more or less indecipherable to anybody who wasn't there with you.

We've gone from a cloudy idea, like "autopay" or "to-do groups," to a specific approach and a handful of concrete elements. But the form we have is still very rough and mostly in outline.

What we've done is landed on an approach for how to solve the

problem. But there may be some significant unknowns or things we need to address before we'd consider this safe to hand off to a team to build successfully.

The next step is to do some stress-testing and de-risking. We want to check for holes and challenges that could hinder the project from shipping within the fixed time appetite that we have in mind for it.

After that we'll see how to wrap up the shaped concept into a write-up for pitching.

No conveyor belt

Also keep in mind that, at this stage, we could walk away from the project. We haven't bet on it. We haven't made any commitments or promises about it. What we've done is added value to the raw idea by making it more actionable. We've gotten closer to a good option that we can later lobby for when it's time to allocate resources.



Risks and Rabbit Holes

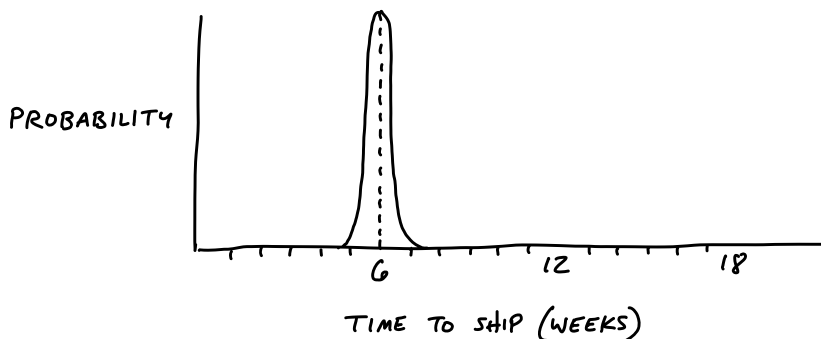
Remember that we're shaping work for a fixed time window. We may trust from our experience that the elements we fleshed out in the previous chapter are buildable within the appetite (six weeks). But we need to look closer, because all it takes is one hole in the concept to derail that. Suppose we bet on the project and a team takes it on. If they run into an unanticipated problem that takes two weeks to solve, they just burned a third of the budget!

Even worse, sometimes you run into problems that don't just delay the project—they have no apparent solution. We once bet on a project to redesign the way we present projects with clients on Basecamp's home screen. We assumed the designer would figure it out; we didn't do the work in the shaping phase to validate that a viable approach existed. Once the project started, it turned out to be a much harder problem than we expected. None of us were able to find a suitable design solution within the six weeks we budgeted. We ended up abandoning the project and rethinking it later.

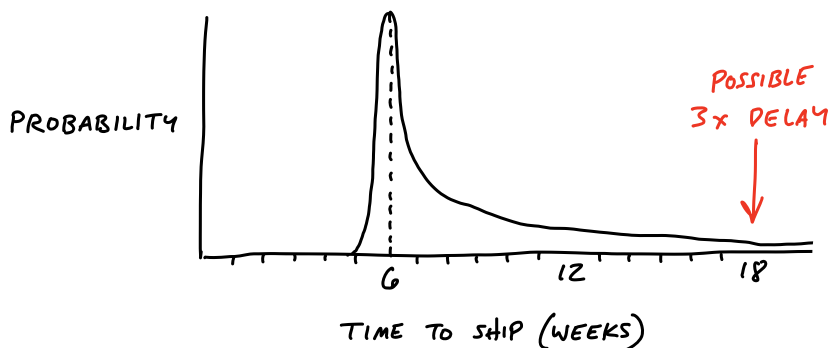
Of course there will always be unknowns. That's why we apply the many practices in Part Three so that teams tackle the right problems in the right order, leaving room for the unexpected. But that doesn't mean we shouldn't look for the pitfalls we *can* find up front and eliminate them before betting on the project. Before we consider it safe to bet on, a shaped project should be as free of holes as possible.

Different categories of risk

In terms of risk, well-shaped work looks like a thin-tailed probability distribution. There's a slight chance it could take an extra week but, beyond that, the elements of the solution are defined enough and familiar enough that there's no reason it should drag on longer than that.



However, if there are any rabbit holes in the shaping—technical unknowns, unsolved design problems, or misunderstood interdependencies—the project could take *multiple times* the original appetite to complete. The right tail stretches out.



We want to remove the unknowns and tricky problems from the project so that our probability is as thin-tailed as possible. That means a project with independent, well-understood parts that assemble together in known ways.

Look for rabbit holes

Fleshing out the elements of the solution was a fast-moving, exploratory process. It was more breadth than depth. In this step, we slow down and look critically at what we came up with. Did we miss anything? Are we making technical assumptions that aren't fair?

One way to analyze the solution is to walk through a use case in slow motion. Given the solution we sketched, how exactly would a user get from the starting point to the end? Slowing down and playing it out can reveal gaps or missing pieces that we need to design.

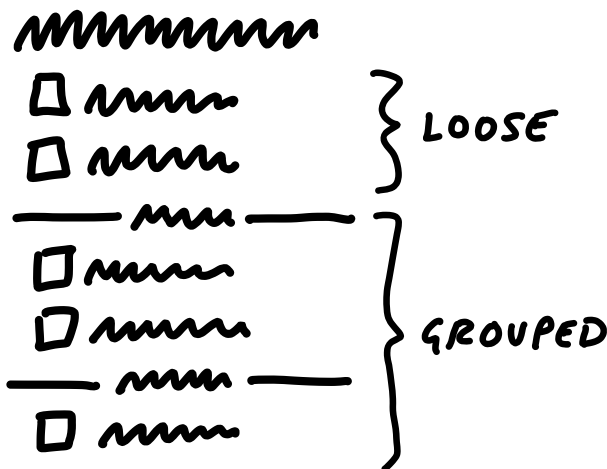
Then we should also question the viability of each part we think we solved. We ask ourselves questions like:

- Does this require new technical work we've never done before?
- Are we making assumptions about how the parts fit together?

- Are we assuming a design solution exists that we couldn't come up with ourselves?
- Is there a hard decision we should settle in advance so it doesn't trip up the team?

Case study: Patching a hole

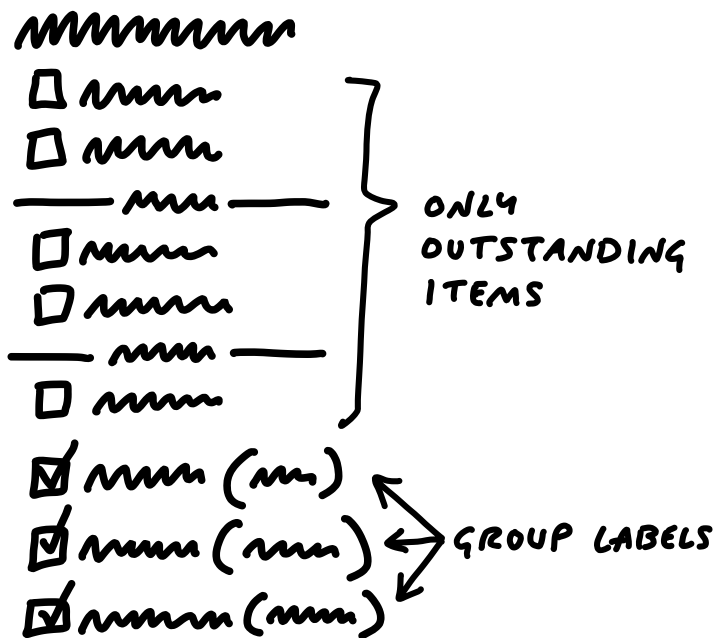
For example, when we defined the To-Do Groups project, we introduced the idea of dividers in the to-do list:



We liked the idea of the dividers, and the logic of loose versus grouped to-dos made sense to us. But when we looked closer we realized that we didn't address how to display completed items. In the pre-existing design, the latest few completed items displayed below the list. Should we now render completed items at the bottom of each group instead of the list? Or should we continue to show completed items at the bottom, and repeat the same set of dividers within the completed items section? Should we reconsider how we handle completed items entirely?

This was a hole in the concept. If we didn't address it, we'd be pushing a deep design problem down to the team and unreasonably asking them to find a solution under deadline. It's not responsible to give the team a tangled knot of interdependencies and then ask them to untangle it within a short fixed time window.

We knew from experience that changing the way completed to-dos render has lots of complicated implications in user experience, navigation, and performance. To remove uncertainty in the project, we decided to dictate a solution in the shaped concept. We would leave the completed items exactly as they worked previously. Instead of grouping or segmenting them, we would just append the name of the group to each completed item. It would be a little messy, but we justified the trade-off: it drastically simplified the problem, and we could still show completed items from a group on the group's detail page.



This is the kind of trade-off that's difficult to make when you're working inside the cycle under pressure. There are lots of reasons why a different design or a deeper reconsideration of completed to-dos would be objectively better. Why not try rendering them inside each group? A designer could reasonably think, "Maybe if I experiment with the styling a little more I can make them blend in better." They could easily waste a few days of the very few weeks they have going down a dead end.

As shapers, we're thinking less about the ultimate design and more about basic quality and risk. With the compromised concept we get to keep all the elements that made the project worth doing—the groups of incomplete items—and we get to cut off a big tail of risk.

Next, when we write the pitch for this project, we'll point out this specific "patch" as part of the concept. That way nobody down the line will get tripped up on it.

Declare out of bounds

Since everyone on the team wants to do their best work, they will of course look for all the use cases to cover and consider them necessary. As the team gets more comfortable with scope hammering (see *Decide When to Stop*), this improves. But it's still a good idea to call out any cases you specifically *aren't* supporting to keep the project well within the appetite.

For example, we worked on an idea for notifying groups of people in Basecamp. Rather than checking off five programmers one by one, you could just click "Programmers" and they'd be selected for notification. As we looked at the product, we saw tons of places where this kind of behavior might make sense. If we let you choose a group when posting a message, why not when assigning a to-do,

or mentioning people in the chat room?

We decided for the purpose of the project that the core value was narrowing down who to notify about a message. We explicitly marked off the other cases as “out of bounds” for the project and focused on the win we wanted: a faster flow for posting messages.

Cut back

There may be parts of the solution we got excited about during the sketching phase that aren’t really necessary. When we designed the To-Do Groups feature, we thought it would be great to color-code groups. No doubt the page would look more interesting with color-coded group labels, and the feature might be more useful too. But we decided to flag this as unnecessary and cut it from the core of the project. We could mention it to the team as a nice-to-have, but everyone should start from the assumption that the feature is valuable without it.

Present to technical experts

Up to this point shaping has been a closed-door activity. Before you’re ready to write up the idea to share more widely, you might need input on some parts of the concept you aren’t completely sure about. There may be a technical assumption that you need to verify with someone who understands the code better. Or perhaps you want to make sure that usage data doesn’t contradict an assumption you’re making about current customer behavior.

This is a good time to grab some technical experts and walk them through the idea. Communicate that this is just an idea. It’s something you’re shaping as a potential bet, not something that’s coming down the pipe yet. The mood is friendly-conspiratorial:

“Here’s something I’m thinking about... but I’m not ready to show anybody yet... what do you think?”

Beware the simple question: “Is this possible?” In software, everything is possible but nothing is free. We want to find out if it’s possible within the appetite we’re shaping for. Instead of asking “is it possible to do X?” ask “is X possible in 6-weeks?” That’s a very different question.

Talk through the constraints of how this is a good solution given the appetite, so they’re partners in keeping the project at the size you intend. And emphasize that you’re looking for risks that could blow up the project. It’s not just a “what do you think” conversation—we’re really hunting for time bombs that might blow up the project once it’s committed to a team.

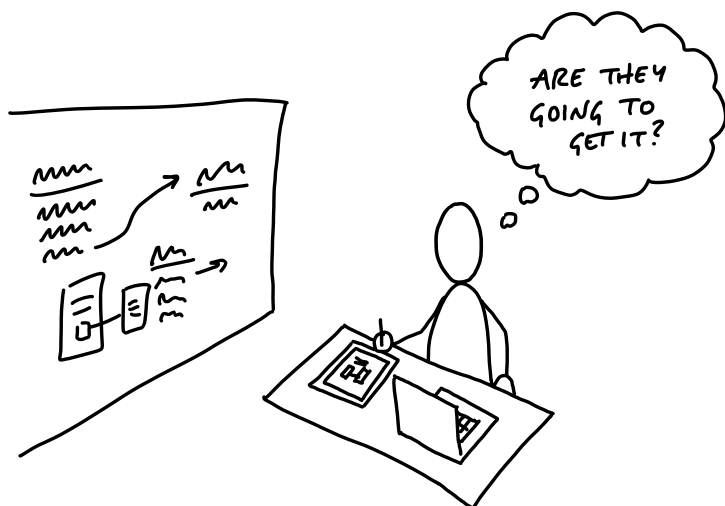
Try to keep the clay wet. Rather than writing up a document or creating a slideshow, invite them to a whiteboard and redraw the elements as you worked them out earlier, building up the concept from the beginning. Stick completely to the concept you already worked out to get feedback on the work you’ve already done. Then once you’ve covered the work you already did, open it up and invite them to suggest revisions. Having seen this concept, do they have any insights about how to drastically simplify or approach the problem differently?

Depending on how the conversation goes, you may either have validated your approach or discovered some problems that send you back for another round of shaping.

De-risked and ready to write up

At the end of this stage, we have the elements of the solution, patches for potential rabbit holes, and fences around areas we've declared out of bounds. We've gone from a roughly formed solution with potential risk in it to a solid idea that we now hope to bet on in the future.

That means we're ready to make the transition from privately shaping and getting feedback from an inner-circle to presenting the idea at the betting table. To do that, we write it up in a form that communicates the boundaries and spells out the solution so that people with less context will be able to understand and evaluate it. This "pitch" will be the document that we use to lobby for resources, collect wider feedback if necessary, or simply capture the idea for when the time is more ripe in the future.



Write the Pitch

We've got the elements of a solution now, and we've de-risked our concept to the point that we're confident it's a good option to give a team. But the concept is still in our heads or in some hard-to-decipher drawings on the whiteboard or our notebook. Now we need to put the concept into a form that other people will be able to understand, digest, and respond to.

This is where we say "Okay, this is ready to write up as a pitch." In this chapter, we'll walk through the ingredients of a pitch and show some fully worked out examples from real projects at Basecamp.

The purpose of the pitch is to present a good potential bet. It's basically a presentation. The ingredients are all the things that we need to both capture the work done so far and present it in a form that will enable the people who schedule projects to make an informed bet.

There are five ingredients that we always want to include in a pitch:

1. **Problem** — The raw idea, a use case, or something we've seen that motivates us to work on this
2. **Appetite** — How much time we want to spend and how that constrains the solution
3. **Solution** — The core elements we came up with, presented in a form that's easy for people to immediately understand
4. **Rabbit holes** — Details about the solution worth calling out to avoid problems
5. **No-gos** — Anything specifically excluded from the concept: functionality or use cases we intentionally aren't covering to fit the appetite or make the problem tractable

Ingredient 1. Problem

It's critical to always present both a problem and a solution together. It sounds like an obvious point but it's surprising how often teams, our own included, jump to a solution with the assumption that it's obvious why it's a good idea to build this thing.

Diving straight into “what to build”—the solution—is dangerous. You don't establish any basis for discussing whether this solution is good or bad without a problem. “Add tabs to the iPad app” might be attractive to UI designers, but what's to prevent the discussion from devolving into a long debate about different UI approaches? Without a specific problem, there's no test of fitness to judge whether one solution is better than the other.

Establishing the problem also lets us have a clearer conversation later when it's time to pitch the idea or bet on it. The solution might

be perfect, but what if the problem only happens to customers who are known to be a poor fit to the product? We could spend six weeks on an ingenious solution that only benefits a small percentage of customers known to have low retention. We want to be able to separate out that discussion about the demand so we don't spend time on a good solution that doesn't benefit the right people.

How far you have to go to spell out the problem will depend on how much context you share with the people reading the write-up. The best problem definition consists of a single specific story that shows why the status quo doesn't work. This gives you a baseline to test fitness against. People will be able to weigh the solution against this specific problem—or other solutions if a debate ensues—and judge whether or not that story has a better outcome with the new solution swapped in.

Ingredient 2. Appetite

You can think of the appetite as another part of the problem definition. Not only do we want to solve this use case, we want to come up with a way to do it in six weeks, not three months, or—in the case of a small batch project—two weeks, not the whole six weeks.

Stating the appetite in the pitch prevents unproductive conversations. There's always a better solution. The question is, if we only care enough to spend two weeks on this now, how does *this specific solution* look?

Anybody can suggest expensive and complicated solutions. It takes work and design insight to get to a simple idea that fits in a small time box. Stating the appetite and embracing it as a constraint turns everyone into a partner in that process.

Ingredient 3. Solution

Like solutions with no problems, sometimes companies bet on problems with no solution. “We really need to make it easier to find things on the messages section. Customers are complaining about it.”

That’s not ready to pitch or bet on. A problem without a solution is unshaped work. Giving it to a team means pushing research and exploration down to the wrong level, where the skillsets, time limit, and risk profile (thin vs. heavy tailed) are all misaligned.

If the solution isn’t there, someone should go back and do the shaping work on the shaping track. It’s only ready to bet on when problem, appetite, and solution come together. Then you can scrutinize the fit between problem and solution and judge whether it’s a good bet or not.

Help them see it

During the elements phase, it was critical to sketch ideas at the right level of abstraction so we didn’t slow down or lose any of the ideas appearing at the corners of our brains and tips of our tongues.

We also need to draw at the right level of detail when we write the pitch. Here the challenge is a little different. We have time to slow down and prepare a proper presentation. We need to stay high level, but add a little more concreteness than when we worked alone or with a partner. People who read the pitch and look at the drawings without much context need to “get” the idea.

We need more concreteness, but we don’t want to over-specify the design with wireframes or high-fidelity mocks. They’ll box in the designers who do the work later. We also risk side-tracking the

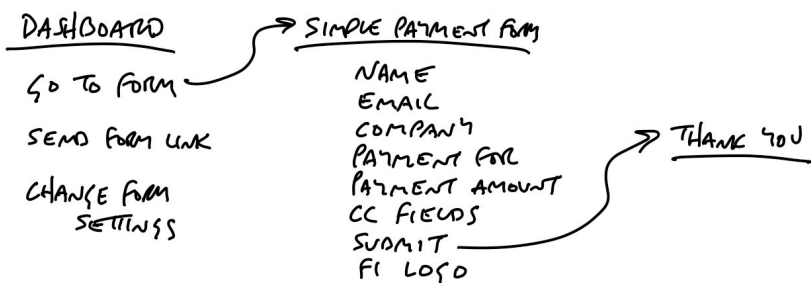
discussion into topics like color, proportions, or layout that have nothing to do with the actual shaping work we did.

At the same time, hand-written breadboards have a “you had to be there” quality to them. To people who didn’t watch the breadboard unfold step by step, it can look like a soup of words and arrows.

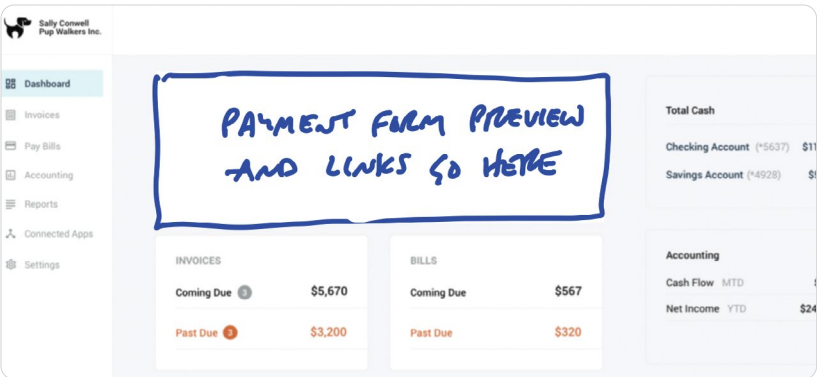
Therefore we need some techniques to help people see the idea while still not going too far into irrelevant details.

Embedded sketches

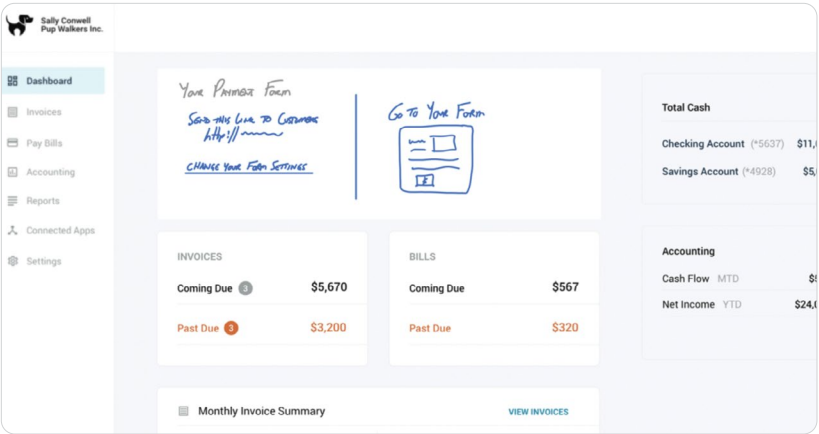
Suppose your breadboard from the shaping session looked like this:



People might have trouble visualizing where these new affordances go on the Dashboard. We could sketch a new box on the Dashboard to make it clearer:



But we’re still asking people to imagine too much. It’s worth the trade-off to go one step down into fat-marker detail here.



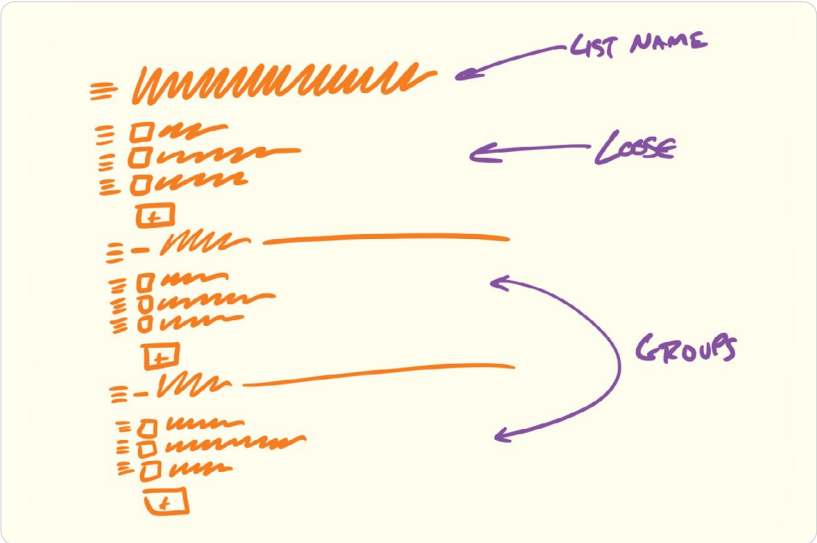
This makes it easier to see what the elements are and evaluate how clearly the feature presents itself on the dashboard. The downside is we’ve gotten into some layout decisions that would have been nice to avoid. Designers should feel free to find a different design than the box divided with a vertical line. We’d add a disclaimer here in the pitch that reminds designers of the latitude they should take.

This is an example of selectively getting into more visual detail because we need it to sell the concept. Fortunately, we won’t need to make as many visual decisions in other parts of the concept. This was a “linchpin” part of the design that everybody had to see concretely in order to “get” it.

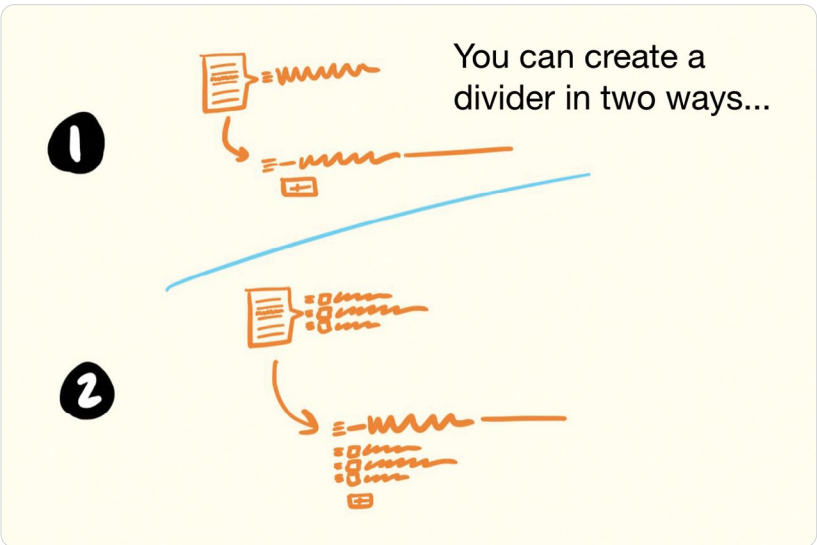
Annotated fat marker sketches

Sometimes ideas are inherently visual or a little too complicated to express in a schematic breadboard. Fat marker sketches can be very effective in a pitch; you just need to take more care to label them cleanly.

Redrawing the sketch on an iPad—still with a fat brush size—works well. You can use different colors to separate the labels from the material parts of the sketch.



Or you might add some call-outs to enable discussion of specific elements.



Ingredient 4. Rabbit holes

Sometimes addressing a rabbit hole just requires a few lines of text. For example, in the Payment Form project above, the shapers wanted to call out a specific solution for how to create URLs. The URLs would never live on custom domains for v1 of the project. This is the kind of thing that's not central to the concept, but spelling it out patches a potential rabbit hole.

Ingredient 5. No Gos

Lastly if there's anything we're *not* doing in this concept, it's good to mention it here. In the case of the Payment Form project, the team decided up front that they wouldn't allow any kind of WYSIWYG editing of the form. Users would only be able to provide a logo and customize the header text on a separate "customize" page. WYSIWYG might be better in some peoples' eyes, but given the appetite it was important to mark this as a no-go.

Examples

Here are two examples of real pitches.

This pitch for grouping to-dos together starts by showing a work-around people are using in the current design. Then it sketches out all the main ideas for how to enable optional to-do groupings.

Grouping to-dos

Jason Fried
Feb 10, 2017 · Skiffed 3 people

For 13 years we've done stuff like this...

OC...

What we're trying to do is create simple groups of to-dos within a single list. So we're hacking it. We're either creating "Arbitrary Division Templates" or pregrouping a batch of to-dos with a label like "QA" or our old standby, the "trusty" which is code for "trusty". Trust it works, but it's messy, and it's not a refined pattern to share with customers. 13 years in, it's time to level up.

To-do groups

A couple weeks ago, Ryan and I spent an afternoon working on an idea not to make to-do groups first class citizens without adding noticeable complexity to the purity of today's to-do lists. We're pleased with where we landed, and we'd like to put it forward as a big beach project this cycle.

Here's how it works:

First, to-dos without groups will look identical to today's to-dos. We won't add any additional UI around them, no new buttons, etc...

But, if you add a group (or groups), it'll look like this...

It's not as messy as the to-do list in the group of to-dos. (2014) - View full-size · Download

This means now you could make a list for a specific scope that had a section for design-specific to-dos, and QA-specific to-dos.

A group is defined by a divider. You can name a divider anything you want, it can't be checked off, it's not a to-do itself. Any to-dos you put below the divider become part of that group. And critically, groups always go below any ungrouped (loose) to-dos. This simplifies a bunch of behaviors, and prevents loose to-dos from being lost between groups. If it's loose, it's at the top.

You can create a divider in two ways...

Key to the concept is that we'll add a 'to-do' button to the bottom of each group (and the loose to-dos at the top). So once a divider exists, you can add to-dos directly to that section. If we didn't do this, and we only had one add-to-do button at the top or bottom, we'd have to add and then move into place. That's a hassle. It's much better to add it place where we have places to add.

Committed to-dos will still be grouped into a single collection at the bottom of the entire list. If they were part of a group when you checked them off, we'll prepand the group name before the to-do like so...

If you uncheck a committed to-do from a group, it could go back up to the group (assuming the group divider hasn't been deleted). (2014) - View full-size · Download

Groups will also have their own permas. So you can click the title of a group/divider, and you'll see a new page with just the to-dos in that group...

Group permas for QA - (2014) - View full-size · Download

What we like about this concept overall is that it's very straightforward. It doesn't change existing to-do lists at all. No new permanent UI in the way, no behavior changes if you don't use groups.

But if you want to level up your organization and we'll use the shit out of this - then you can add groups to a list. No groups within groups, no indenting - just as many one-level groups as you'd like.

There are open questions. Things like if you move the divider, do all the items move with it? If yes since it's a group, but we have to work that out. Also, since you can't put groups above loose to-dos, we'll have to prevent dragging above a certain point. But that's doable as well. I'm sure there are a few other questions as well, but we can work those out as we go.

SKETCH: Down the road we'd like to explore adding to-do templates to BCD. We have project templates, but now we're just talking to-do templates. You can imagine creating a template with no to-dos, but with groups in place. This is a process based - people can organize projects in similar ways with just a little bit of structure like this...


Just add to place - (2014) - View full-size · Download

Two screenshots demonstrate the problem. Fat marker sketches describe the solution. Rabbit holes motivated some of the sketches.

See full size at basecamp.com/shapeup-todo-pitch.

This pitch for changing how notifications work starts with two videos to demonstrate the problem. The black boxes toward the end are a visualization of user behavior data that supports a decision in the pitch.

Groups, clarified

 Pitch by Ryan Singer
Feb 16, 2019 • Notified 53 people

Last November I wrote a pitch for adding Groups via RCM to RCS. [@Group_Product](#) [@Bacon](#)

After looking at the pitch, [@Bacon](#) pointed out it wasn't entirely clear where to expect Groups to appear. Example: Suppose you could autocomplete a Group when you @mention somebody. Does that set an expectation you use Groups in other places where you autocomplete people, like when assigning to-do's?

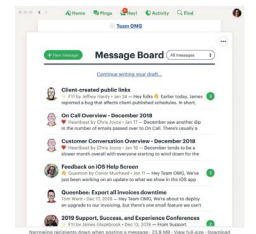
We could answer "yes" and expect Groups everywhere we can think of. But that increases the scope. And it's a hard product design email to add functionality for "autocomplete" sake. It's better to understand use cases and target them.

With our last RCS cycle approaching, I wanted to take one last swing at this. Here's a narrower use case and a narrower solution that I think sets clear expectations about where and how Groups should appear in the app.

Narrowing down a problem case

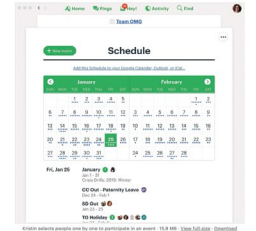
To find a more specific use case to target, I turned to [@Krisde](#) for perspective. She and others in Kotlin have wanted Groups for a long time.

I asked Kotlin to demonstrate workflows she's already doing today that are painful without Groups. The aim was to use some concrete "before" cases to design against. She had two really good ones. Both of them involve **narrowing down** 50 people to 15. Here's a video of Kotlin running around to check off support people when posting a Message:



The screenshot shows the 'Message Board' interface with a list of messages. A modal titled 'Who should be notified?' is open, showing a list of names with checkboxes. The modal is annotated with a red arrow pointing to the 'Who should be notified?' title and a red arrow pointing to the 'Who should be notified?' title.

The same problem comes up when choosing who an event is "with." Here she has to pick 10 people out by one using the autocomplete UI.



The screenshot shows the 'Schedule' interface with a calendar view. A modal titled 'Who should be notified?' is open, showing a list of names with checkboxes. The modal is annotated with a red arrow pointing to the 'Who should be notified?' title and a red arrow pointing to the 'Who should be notified?' title.

Watching these videos, you see a palpable pain. With this as the "before", it's clear how Groups could offer a better "after" in this use case.

@Mentions: Cool, but maybe not calm enough

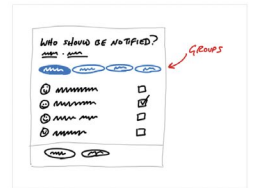
After this, I asked: can we find a similar pain point involving @mentions? We couldn't come up with one as compelling.

It's easy to think of cases where mentioning everybody via a Group could be handy. But is that good for our way of life or how we communicate with each other? With 15 people behind a single dimension, it would be easy to spam each other without thinking too hard about it.

With that in mind, I looked for broad design solutions for those two use cases only.

Solution for choosing Message subscribers

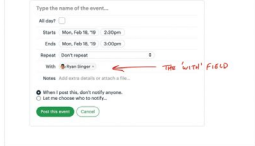
We could use a RCS-style approach to solve the first case. We have a single modal today for setting Subscribers separately, whether you're posting a new Message or changing who gets notified about some commentable. That "Who should be notified?" modal could offer Groups at the top. Clicking them checks members of the Group below.



The sketch shows a modal titled 'Who should be notified?'. It has a list of names with checkboxes. A red arrow points to the 'Groups' section, which contains a list of group names with checkboxes.

Do Events get a different solution?

Solving the Events "with" field case is not as straightforward. The current UI is a single autocomplete field.



The sketch shows a modal titled 'Who should be notified?'. It has a list of names with checkboxes. A red arrow points to the 'Who should be notified?' section, which contains a list of names with checkboxes.

We could offer Groups inside that autocomplete, but a whole bunch of problems follow. We don't want to set the precedent that a Group might appear inside any autocomplete, per IP's concerns in the intro above. And if we did... there are still UX problems. Suppose you choose a Group in the autocomplete. Would the Group name appear then in the With field? That'd be misleading because maybe only 10 people from the Group are on the project. We could insert the people from the Group into the autocomplete field instead of the Group... but then it'd tend to use which Groups you already added. So autocompleteable Groups are out.

Next idea. What if we replaced the autocomplete with a modal picker as "Who should be notified?" Then we could re-use the same solution as messages.

Technically that could work. Some "Choose participants..." button would launch the modal and then populate the field.

However, this could be a UX step backwards from today. Suppose the vast majority of people are autocompleteing one or two names. That's a very different behavior from choosing Message subscribers, with everyone checked by default. Depending on the number of people on the project, you might have to scroll through a few screens in the modal to check the one other name you want. It's easier to autocomplete.

I took a look at the data for perspective. Indeed, the majority of Events have one or two participants. And the pattern is the opposite for Message subscribers. Here's a peek (explaining the analysis would take another write-up).



The data also showed that our use case — choosing 10 people in the With field — is extremely rare. So we don't want to rethink the UI just for this one.

Workaround: This is only about Subscribers

I came back to Kotlin with a proposal: if she had to stop using the "With" field and instead relied on the "Who should be notified?" instead, would that still work for her when she posts events?

The self's would be fine for us. For others, it could conceivably be a problem because setting Subscribers vs. Participants sort the same — Participants gets a calendar integration that Subscriber doesn't. But in our case — and we're the outliers here — it's not a problem.

This suggests we can confine the Groups feature entirely to "Who should be notified?" — the Subscriber modal.

Proposed solution

We could define these "Subscriber groups" in Android, as pitched previously:



At a minimum, we could confine the new functionality to that Subscribers modal.

If we want to, we could expand to also invite people to projects with the same approach. The models are nearly identical. It's stretch for that is on the [pitch](#) path.

What we're not doing

This goes a step further narrower than the last pitch. No autocomplete Groups anywhere. Still no connection with Teams, just a better way to narrow down Subscribers any place in the app that we let you choose Subscribers via the "Who should be notified?" modal.

Last chance for this one in the next cycle! Hope this sharpened it down enough.

Two videos show the problem. A fat marker sketch and a breadboard describe the solution. The black boxes contain data visualizations that support trade-offs in the solution. See full size at basecamp.com/shapeup-groups-pitch.

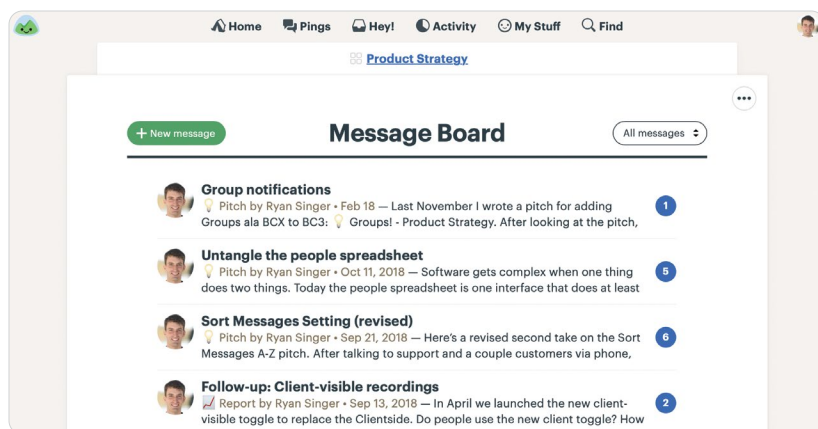
Ready to present

The next step will be to make the case that this pitch describes a bet worth making. This can happen in a couple ways.

We prefer asynchronous communication by default and escalate to real-time only when necessary. This gives everyone the maximum amount of time under their own control for doing real work. That means the first step for presenting a pitch is posting the write-up with all the ingredients above somewhere that stakeholders can read it on their own time. This keeps the betting table short and productive. In ideal conditions everyone has time to read the pitches in advance. And if that isn't possible in some cases, the pitch is ready to pull up for a quick live sell.

How we do it in Basecamp

We post pitches as Messages in Basecamp. We created a Message Category called *Pitch* so we can easily find them. Pitches are posted to a Team called *Product Strategy* that can be accessed by people on the betting table.



Pitches on the Message Board of the Product Strategy team in Basecamp