

---

## **Shape Up - Stop Running in Circles and Ship Work that Matters**

Reading Notes by Vasken Dermardiros

Ryan Singer from BaseCamp

Shape Up is for product development teams who struggle to ship. If you've thought to yourself "Why can't we ship like we used to?" or "I never have enough time to think about strategy," then this book can help. You'll learn language and techniques to define focused projects, address unknowns, and increase collaboration and engagement within your team. (Book can be found here: <https://basecamp.com/shapeup>)

## Contents

<b>Summary</b>	<b>4</b>
<b>Chapter 1: Introduction</b>	<b>6</b>
Growing Pains . . . . .	6
Work in Six-Week Cycles . . . . .	6
Shaping the Work . . . . .	6
Making Teams Responsible . . . . .	7
Targeting Risk . . . . .	7
<b>Chapter 2: Principles of Shaping</b>	<b>7</b>
Wireframes are too Concrete . . . . .	7
Words are too Abstract . . . . .	7
Properties of Shaped Work . . . . .	8
Who Shapes . . . . .	8
Two Tracks . . . . .	8
Steps to Shaping . . . . .	9
<b>Chapter 3: Set Boundaries</b>	<b>9</b>
Setting the Appetite . . . . .	9
Fixed Time, Variable Scope . . . . .	9
Responding to Raw Ideas . . . . .	10
Narrow Down the Problem . . . . .	10
Watch out for Grab-bags . . . . .	10
<b>Chapter 4: Find the Elements</b>	<b>10</b>
Move at the Right Speed . . . . .	10
Fat Marker Sketches . . . . .	11
Not Deliverable yet and can also be Discarded . . . . .	11
<b>Chapter 5: Risks and Rabbit Holes</b>	<b>11</b>
Present to Technical Experts . . . . .	11

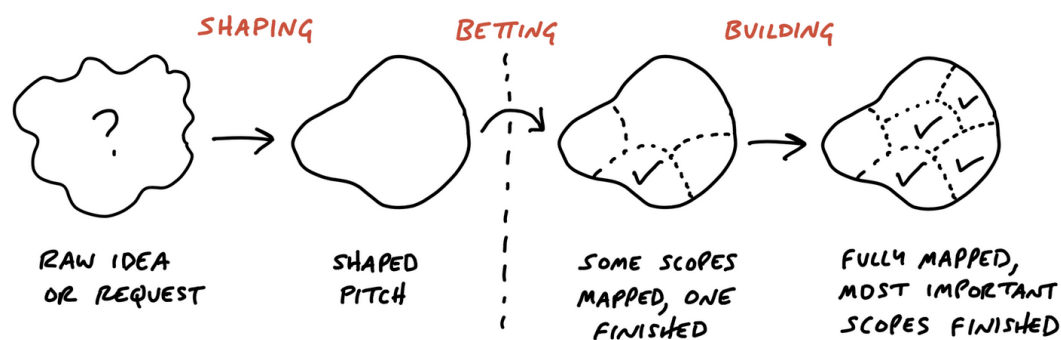
De-Risked and Ready to Write Up . . . . .	12
<b>Chapter 6: Write the Pitch</b>	<b>12</b>
Ingredient 1. Problem . . . . .	12
Ingredient 2. Appetite . . . . .	13
Ingredient 3. Solution . . . . .	13
Help them see it . . . . .	13
Ingredient 4. Rabbit Holes . . . . .	13
Ingredient 5. No Gos . . . . .	13
Ready to Present . . . . .	13
<b>Chapter 7: Bets, Not Backlogs</b>	<b>14</b>
No Backlogs . . . . .	14
A Few Potential Bets . . . . .	14
Decentralized Lists . . . . .	14
Important Ideas come Back . . . . .	14
<b>The Betting Table</b>	<b>15</b>
Six-week Cycles . . . . .	15
Cool-down . . . . .	15
Team and Project Sizes . . . . .	15
The Betting Table . . . . .	15
The Meaning of a Bet . . . . .	16
Uninterrupted Time . . . . .	16
The Circuit Breaker . . . . .	16
What about Bugs? . . . . .	16
Keep the Slate Clean . . . . .	17
<b>Chapter 9: Place Your Bets</b>	<b>17</b>
R&D Mode . . . . .	17
Production Mode . . . . .	18
Cleanup Mode . . . . .	18
<b>Chapter 10: Hand Over Responsibility</b>	<b>18</b>
Assign projects, not tasks . . . . .	18
Done means Deployed . . . . .	19
Kick-off . . . . .	19
Getting Oriented . . . . .	19
Imagined vs Discovered Tasks . . . . .	19

<b>Chapter 11: Get One Piece Done</b>	<b>19</b>
Integrate One Slice . . . . .	19
Programmers don't need to Wait . . . . .	20
<b>Chapter 12: Map the Scopes</b>	<b>21</b>
Organize by Structure, not by Person . . . . .	21
The Scope Map . . . . .	21
The Language of the Project . . . . .	21
Discovering Scopes . . . . .	24
How to know if the Scopes are Right . . . . .	24
Chowder . . . . .	24
Mark nice-to-haves with ~ . . . . .	25
<b>Chapter 13: Show Progress</b>	<b>25</b>
Estimates don't show Uncertainty . . . . .	25
Work is like a Hill . . . . .	25
Scopes on the Hill . . . . .	26
Nobody says "I don't know" . . . . .	28
Build your way uphill . . . . .	28
Solve in the right Sequence . . . . .	28
<b>Chapter 14: Decide When to Stop</b>	<b>28</b>
Scope Hammering . . . . .	28
QA is for the Edges . . . . .	29
When to Extend a Project . . . . .	29
<b>Chapter 15: Move On</b>	<b>30</b>
Let the Storm Pass . . . . .	30
Stay Debt Free . . . . .	30
<b>Conclusion</b>	<b>30</b>

## Summary

The idea. Work is formulated into rigid six week cycles in smaller teams that are defined loosely but with strong boundaries. Because of the strong time deadline, projects don't drag on but need to have a realistic appetite, and because there is no backlog, the work can be adapted to evolving situations without having to be extremely reactive.

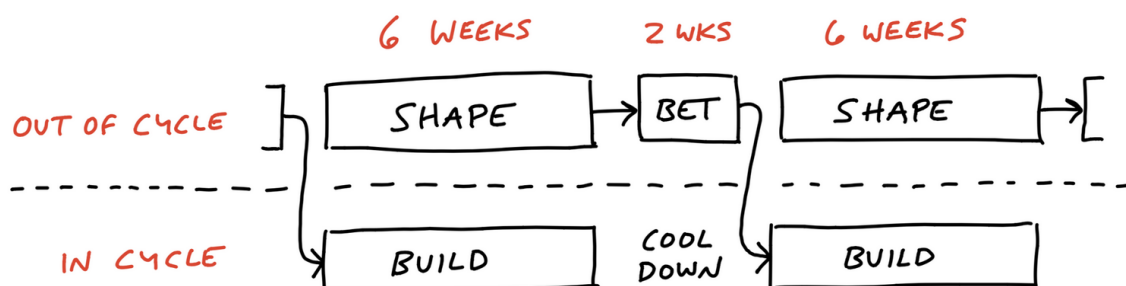
The process. Raw and ambiguous ideas are shaped into projects. The projects are pitched on the betting table where upper management accepts, rejects or proposes changes to the project (the pitch) and then it gets handed to the ones building it out. The building team typically is a designer with one or two programmers. The designer can be a domain expert, a UX person, etc.



*The phases of the work still hold true even if you don't work in cycles or have dedicated people to do the shaping and building*

**Figure 1:** The phases of shaping and delivering

While the team is building the project, the next cycle is shaped in parallel.



*With more people, shaping and building happen on separate tracks and bets are made to fill six-week cycles*

**Figure 2:** Out and In cycle tracks

After the six weeks and the project delivered and in production, the development team enjoys a cool-down period where they can turn their focus to bug fixes, documentation, reading and so on. During

this time, the next cycle projects are bet on. Note that there is no backlog. If a project is not retained, it is discarded. If the idea is good, the person who shaped it will need to pitch it again in the next cycle *with updated information*.

## Chapter 1: Introduction

Instead of a manager breaking the work and handing tasks to programmers and then managing them, the idea is to have senior people *shape* a six-week (no extensions) project as well as possible then *bet* with programmers on which aspects they'd like to have in it, and then the programmers are free to work however they want. Senior people can move on to shape another project.

### Growing Pains

Common struggles:

- Team members feel like projects go on and on, with no end in sight.
- Product managers can't find time to think strategically about the product.
- Founders ask themselves: "Why can't we get features out the door like we used to in the early days?"

### Work in Six-Week Cycles

Six weeks is long enough to build something meaningful start-to-finish and short enough that everyone can feel the deadline looming from the start, so they use the time wisely. The majority of our new features are built and released in one six-week cycle.

No daily meetings. No counting hours. No changing direction every two weeks. Focus at a higher level. "If this ships in six weeks, we'll be really happy. We'll feel our time was well spent."

### Shaping the Work

Small senior group defines the key elements of a solution before betting on it. They keep it *abstract enough* to keep room to work out the interesting details.

When *shaping*, focus is more on **appetite**: how much time do we *feel like spending* on this vs. how much time will it *take*? Design the solution to fit on the appetite. How much do your clients consume the work? Matching the appetites for consuming and developing.

## **Making Teams Responsible**

Give **full responsibility** to a small integrated team of designers and programmers. They define their own work, tasks, etc.

Conventional approach: managers chopping up work and letting programmers act like ticket-takers.

Senior people can instead work on shaping up better projects. Better shaped projects have clearer boundaries and allow teams to then work more autonomously.

## **Targeting Risk**

We reduce risk in the shaping process by solving open questions *before* we commit the project to a time box. We don't give a project to a team that still has rabbit holes or tangled interdependencies.

Risk of time wastage is handled with the six week period. By default, no extension is given.

Projects need to stand end-to-end. No risk of hoping they'll fit at the end.

## **Chapter 2: Principles of Shaping**

Getting to the right level of abstraction: not too vague and not too concrete. The shapers shape a project the builders will work on while they themselves shape the future work, alone and quickly. Well shaped projects will limit time wasted and provide clear boundaries while maintaining liberty to dream up a solution within that space.

### **Wireframes are too Concrete**

Too much of it is defined, it's basically a design and so it's hard to un-think it and then it's hard to estimate time since matching to the wireframe can have hidden complexities.

### **Words are too Abstract**

Too vague and people won't know what to do. Almost like saying "we'll know it when we see it". More concrete needs will create a boundary for the project and constrain it from growing out of control.

## Properties of Shaped Work

(They give the example of the Dot Grid feature.)

- Property 1: It's rough
  - Looking at it and you know it's unfinished.
- Property 2: It's solved
  - It's been thought through. All the main elements of the solution are there at the macro level and they connect together.
- Property 3: It's bounded
  - Tells you what *not* to do. Where to stop and there's a specific appetite.

## 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.

You don't need to be a programmer but you need to be technically literate.

You have to be critical: what are you trying to solve? Why does it matter? What counts as success/failure? What's the cost of doing this instead of something else?

You mainly work alone, closed-door. You can have a collaborator, you move fast, speak frankly and jump from one promising position to another.

## Two Tracks

You can't schedule shaping work because it's unshaped, risky and unknown.

Two tracks: one for shaping, one for building.

While the builders are building for six weeks, the shapers are shaping potentially what will be built next. **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.**



## Steps to Shaping

Four main steps:

1. **Set boundaries**
2. **Rough out the elements:** sketch the solution without the fine details.
3. **Address risks and rabbit holes:** look for unanswered questions, cut things out, specify details at tricky spots to prevent getting stuck or wasting time.
4. **Write the pitch:** make it into 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.

## Chapter 3: Set Boundaries

Get to the bottom of the ask to not waste time. Projects have a fixed time which ends up prioritizing certain things over others, otherwise things would last forever. The amount of people on a project can vary. Take suggestions/ideas but don't promise anything.

### Setting the Appetite

Stay focused and think how valuable the idea is. Do we want to work on it? Is it a customer ask? Will it take a whole session or is it 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.

### 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.

Without a deadline, you can't make a trade-off. If you have one week left, what would you prioritize? Leave out?

## Responding to Raw Ideas

“Interesting. Maybe some day.” → basically a very soft “no”. 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.

## Narrow Down the Problem

Even if the client asks for something very complicated, ask questions. Dig. There could be a one-day change instead of a six-week project.

Flip the question: “what could we build?” → “what’s really going wrong?” What’s driving the request?

## Watch out for Grab-bags

Worst offenders are “redesigns” or “refactorings” that aren’t driven by a single problem or use case. Ask “what’s not working?” to figure out which parts can stay the same and what parts need to change.

## Chapter 4: Find the Elements

Rough out the idea to prepare it to go through stress-tests and to de-risk. The idea is not safe until that’s done and can still be discarded at this point.

## Move at the Right Speed

Work mostly alone here or with someone else who can keep up and be frank.

- 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?

The simily to use is the breadboard. It has enough to see the logic work but it’s far from being a product.

They show an example on how to set up an invoice system, gradually adding details and playing it out. They then realize there are some ambiguities in the use and rethink the logical order.

## **Fat Marker Sketches**

By using fat markers, details are inherently hard to add. You can only do broad strokes.

When the designers and programmers come in, they see a rough drawing which gives them the opportunity to fill in the details. You don't need to say "sorry but this should've been like this and that".

## **Not Deliverable yet and can also be Discarded**

The ideas so far are very rough and indecipherable to anyone else who wasn't there.

The form is not considered safe to hand off since it has to go through stress-testing and de-risking first. Need to check for holes and challenges to make sure it fits in the 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.

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.

## **Chapter 5: Risks and Rabbit Holes**

Think of the risks in a project and find solutions for them. If unsure, consult technical experts to show them your idea so far and ask if it can be done in six weeks and whether there are pitfalls and more simpler solutions. Afterwards, you can talk more about extensions. Once done, the project is de-risked and can be formatting into a more formal "pitch" and brought up to the betting table to lobby for resources.

Look for rabbit holes to reduce the risk of a project going above budget – six weeks.

Declare out of bounds. Put up fences.

Cut back on "nice to have features" in the actual shaping. You can mention them as extras to do but not necessary.

## **Present to Technical Experts**

When the project is sufficiently shaped, you may solicitate the input of technical experts on the parts that you aren't too sure about. "Here's something I'm thinking about... but I'm not ready to show anybody yet... what do you think?" "Is this possible?"... "Is this possible in six weeks?"

What are the risks in the project that this can blow up?

Keep the clay wet: invite them to a whiteboard, not a full-on slide show. Make it so that it's easy to switch it up. Focus on the work that is already done before opening it up to other stuff. Having seen this concept, do they have any insights about how to drastically simplify or approach the problem differently?

## 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.

Time to move from privately shaping and getting feedback from the 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.

## Chapter 6: Write the Pitch

When the project is mature enough, it is compiled into a pitch – a presentation. There are five ingredients that we always want to include in a pitch:

- **Problem** — The raw idea, a use case, or something we've seen that motivates us to work on this
- **Appetite** — How much time we want to spend and how that constrains the solution
- **Solution** — The core elements we came up with, presented in a form that's easy for people to immediately understand
- **Rabbit holes** — Details about the solution worth calling out to avoid problems
- **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

Really dangerous to jump straight to a solution (what to build) without the problem. Why is this a good idea? Useful? How do you test for fitness?

What's the point of having a great solution to a infrequent problem?

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.

## Ingredient 2. Appetite

Not only do we want to solve the use case, but we need to be able to do it in six weeks (or two weeks for a `small batch` project).

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

No point bringing up a problem without the solution -> this is unshaped work. This is not ready to pitch or bet on!

A valid pitch **must** have all three ingredients!

## Help them see it

Hard for others to “get” what you’re trying to say/convey if they only see a list with arrows.

Make it clearer with sketches.

## Ingredient 4. Rabbit Holes

Could the project lead to something that’s not 100% necessary for v1? Address those upfront.

## Ingredient 5. No Gos

If there are things we’re *not* doing in the concept, it’s good to mention it here.

Remember that there’s a six week budget.

## Ready to Present

The pitch is written up in a shared document that people can read async which helps keep the betting table session short and productive.

People can view it and attach comments, but the “yes” or “no” is reserved for the betting table.

## Chapter 7: Bets, Not Backlogs

Now that we've written a pitch, where does it go? It doesn't go onto a backlog. Ideas are cheap, so if it's important, it'll come back and with new context.

### No Backlogs

A mountain of tasks that will never be done. No.

Backlogs are big time wasters too. The time spent constantly reviewing, grooming and organizing old ideas prevents everyone from moving forward on the timely projects that really matter right now.

### A Few Potential Bets

Before each six-week cycle, hold a **betting table** where stakeholders decide what to do in the next cycle. At the betting table, they look at the pitches from the last six weeks – or any pitches that somebody purposefully revived and lobbied for again.

Nothing else is on the table. No giant list.

If we decide to bet on a pitch, it goes into the next cycle to build. If we don't, we let it go. There's nothing we need to track or hold on to.

What if the pitch was great, but the time just wasn't right? Anyone who wants to advocate for it again simply tracks it independently—their own way—and then lobbies for it six weeks later.

### Decentralized Lists

No backlogs! If things are important, they need to be brought up again and with next context and purpose.

**Everything needs to be fresh.**

### Important Ideas come Back

Ideas are cheap.

Really important ideas will come back.

## The Betting Table

### Six-week Cycles

The whole company follows the six-week cycle: so at the beginning of the cycle, you know everyone is available and ready.

Unlike two-week sprints, six-weeks allows people to think about how to approach the problem and does not require as much overhead for “sprint planning” and doesn’t break the momentum to regroup.

### Cool-down

After the six-week cycle, take a two week cool-down. This is where programmers and designers are free to work on whatever they want, fix bugs, explore new ideas, or try out net technical possibilities.

### Team and Project Sizes

A team is either one designer and two programmers or one designer and one programmer. They’re joined by a QA person who does integration testing later in the cycle.

They either work on one large project **big batch** or multiple small projects **small batch**. Small batch projects run one or two weeks each. They aren’t scheduled individually; it’s up to the small batch team to figure out how to juggle the work so they all ship before the end of the cycle.

### The Betting Table

Held during the cool-down period where stakeholders decide what to do in the next cycle. Potential bets include new pitches shaped during the last six weeks, or possibly one or two older pitches that someone specifically chose to revive. There’s no “grooming” or backlog to organize.

Betting table at Basecamp: CEO, CTO, senior programmer and a product strategist. Call rarely goes longer than an hour or two. Everyone has studied the pitches beforehand.

The highest people in the company are there. There’s no “step two” to validate the plan or get approval. No interference or interruptions later either.

This buy-in from the very top is essential to making the cycles turn properly. The meeting is short, the options well-shaped, and the headcount low. With cycles long enough to make meaningful progress and shaped work that will realistically ship, the betting table gives the C-suite a “hands on the wheel” feeling they haven’t had since the early days.

## The Meaning of a Bet

Bets have a payout. It's not just about filling a time box until it's full. There's something meaningful at the end of it.

Bets are commitments. We're giving the team the entire six weeks to work exclusively on that thing with no interruptions. We're expecting big movements.

A smart bet has a cap on the downside. At worst, we will lose six weeks. No extensions.

## Uninterrupted Time

**It's not really a bet if we say we're dedicating six weeks but then allow a team to get pulled away to work on something else.**

"Just a few hours" or "just one day" -> No! Don't be fooled. Losing the wrong hour can kill a day. Losing a day can kill a week. It kills momentum!

What if something important comes up? We can bet on it when the cycle ends. It'll wait at most six weeks to start getting worked on. That's why it's so important to only bet one cycle ahead.

## The Circuit Breaker

Very few projects are of the "at all costs" type, so in general, if the project can't finish in six weeks, we break it. It means it wasn't shaped well and the appetite wasn't right.

For the next cycle, we can rethink the pitch on how to avoid the rabbit holes. Instead of spending more time on a bad approach, the circuit breaker pushes us to reframe the problem.

This motivates teams to take more ownership over their projects. Give them full responsibility for executing projects. They will ask questions and make hard decisions about where to stop, what to compromise and what to leave out.

## What about Bugs?

What about them?

They are not more important than everything else. It doesn't give an excuse to interrupt ourselves or other people. All software has bugs. Question is: how severe are they? If it's a crisis, sure, drop everything to fix it. But *crises are rare*. You can't ship anything new if you have to fix the whole world first.

When to fix them:



1. **Use cool-down:** two weeks is long enough to fix them.
2. **Bring it to the betting table:** if a bug is too big to fix during cool-down, it can compete for resources at the betting table. There's a huge difference between delaying other work to fix a bug versus deciding up front that the bug is worth the time to fix.
3. **Schedule a bug smash:** once a year – usually around the holidays – we'll dedicate a whole cycle to fixing bugs.

## Keep the Slate Clean

Every cycle starts with a clean slate. Never carry scraps of old work over without first shaping and considering them as a new potential bet.

What if something would take more than a cycle? Envision the feature that fits into one and shape again for the second cycle. Things change. Keep options open to change course.

## Chapter 9: Place Your Bets

- **Existing Products:** Shaped work will occupy the space that's empty around the existing product. Like building a piece of furniture that will fit inside a built house.
- **New Products:** More akin to building where the walls and foundation should go. These tend to take around three cycles, but betting is for one cycle at a time.

## R&D Mode

At the very beginning, we don't know what we want. The work here is a lot more scrapwork to test and try different things to discover what the thing is.

Instead of betting on a well-shaped pitch, we bet the *time* on spiking some key pieces of the new product idea.

The CTO acts as the programmer, the CEO as the designer and come up with the product's future: they define the "holes" where this product will fit.

Nothing is expected to ship at the end of the R&D cycle.

Don't commit to a client for more than one cycle but the R&D cycle doesn't have a product, so nothing is committed after it.

## **Production Mode**

When the R&D phase continues, it'll eventually reach a point where the most important architectural decisions are settled. The product does those few essential things and the foundation is laid for the dozens of other things.

Production mode follows a normal cycle building on an “existing product” which is what was one at the R&D cycle.

Shipping is the goal, not spiking. Since it's not advertised to clients, “shipping” means merging to the main codebase and expecting not to touch it again. Can remove features too afterwards.

## **Cleanup Mode**

Before launching something, we need to make sure everything is good.

Cycle is closer in spirit to the “bug smash”. Leadership stands at the helm throughout the cycle, calling attention to what's important and cutting away distractions.

There aren't clear team boundaries.

Work is “shipped” (merged to the main codebase) continuously in as small bites as possible.

Discipline is still important. “Final cut” decisions. Smaller surface area means less to support and less to commit.

## **Chapter 10: Hand Over Responsibility**

How does the team get started?

### **Assign projects, not tasks**

We *don't* start by assigning tasks to anyone. Nobody plays the role of “taskmaster” or the “architect” who splits the project into small pieces.

Putting things in tasks == putting the project through a paper shredder. No longer whole.

Talented people don't like to be treated like “code monkeys” or ticket takers.

Project was already shaped; boundaries were set; now we trust them to fill in the outline.

## **Done means Deployed**

Within the six week period. QA needs to happen *within* the cycle.

Documentation, marketing material and announcements can come after since those won't entail a 5x time commitment.

## **Kick-off**

Start a new project in the platform with the original pitch or a distilled version of it.

## **Getting Oriented**

The first few days will have no sort of visible work. People are thinking and looking into how to start. Don't interfere, unless it's been three days and there's absolutely nothing.

## **Imagined vs Discovered Tasks**

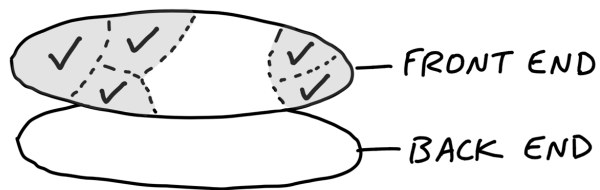
The very first tasks might be imagined already. Later on, while working to implement something, we might discover tasks to work on and links to make. We only ever figure those out while working on the project (and that's why it might be counterproductive to "architect" the whole thing up-front).

## **Chapter 11: Get One Piece Done**

Aim to make something tangible and demoable early – in the first week or so. That requires integrating vertically on one small piece of the project instead of chipping away at the horizontal layers.

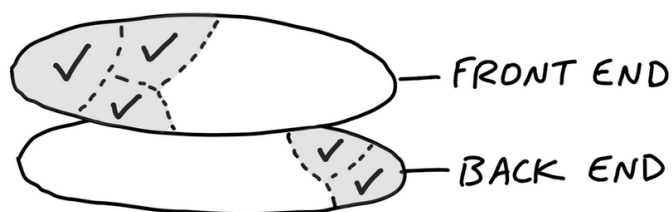
## **Integrate One Slice**

Get a small slice that works -> motivator and a starting point from where to spread.



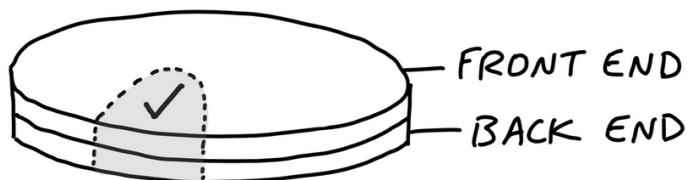
NOTHING WORKS

**Figure 3:** Work on just front-end



NOTHING WORKS

**Figure 4:** Work on a bit of front and a bit of back



SOMETHING WORKS!

**Figure 5:** A small slide that works!

### Programmers don't need to Wait

If there's one small thing that works, programmers can continue expanding it on their end -> if for one button it works, they can imagine the work front-end will do for other buttons and get to work on the back-end.

They don't need to worry about font, colour, the writing, spacing, layout, etc.

Raw. But functional. Program just enough for the next step.

Get working: **core functionality, small amount, novel**. High impact basically and crush uncertainty quickly/early.

## Chapter 12: Map the Scopes

### Organize by Structure, not by Person

When asked to organize tasks for a project, people often separate work by person or role: they'll create a list for Designers and a list for Programmers. This leads to the problem we talked about in the previous chapter — people will complete tasks, but the tasks won't add up to a finished part of the project early enough.

Organize instead based on the *structure* of the project: the things that can be worked on and finished independently of each other. Organizer can see which areas are done and which areas have outstanding work.

We call these integrated slices of the project *scopes*. We break the overall scope (singular) of the project into separate scopes (plural) that can be finished independently of each other. In this chapter, we'll see how the team maps the project into scopes and tackles them one by one.

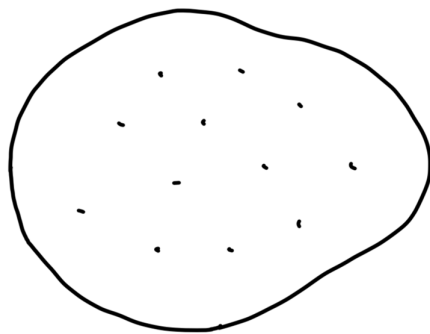
### The Scope Map


Starting from the boundaries of the project, start discovering tasks that go inside. Tasks are concrete and very granular. At this point, it's very hard to see if there are ways of dividing the map into territories that can be completed independently.

### The Language of the Project

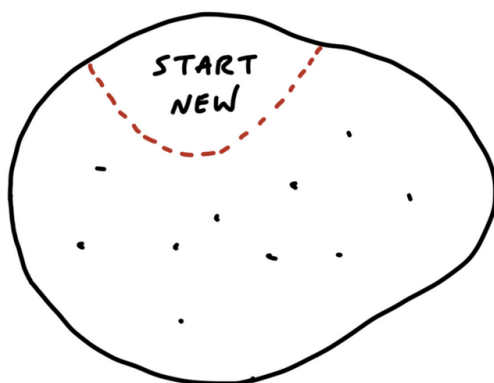
Scopes are more than just slices. They become the language of the project at the macro level. When X is done, we can implement Y then update Z.



When it's time to report a status, the team uses XYZ scopes to communicate. Conversation ends up being at the high-level instead of going through the weeds.



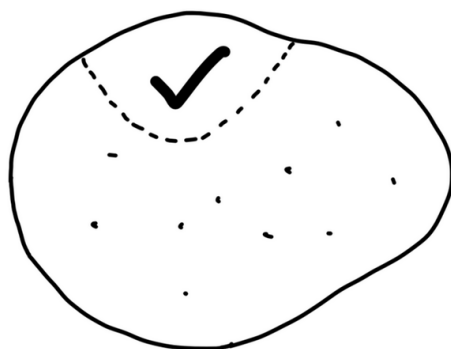
- ≡  **Unscoped**
- ≡ ☐ Intercept attempts to reply to Topic if a draft from a different message exists
- ≡ ☐ Handle draft message timestamps after sending
- ≡ ☐ Hook up Send from Draft edit state
- ≡ ☐ Remember draft content when editing draft
- ≡ ☐ Remember addresses when editing draft
- ≡ ☐ Hook up re-saving from Draft edit state
- ≡ ☐ Reduce duplication in Draft forms
- ≡ ☐ Design index of existing Drafts
- ≡ ☐ Hook up Draft deletion from Draft edit state
- ≡ ☐ Design "Only you can see this draft" wrapper/labels
- ✓ Allow 'invalid' drafts to be created (without an addressee, for example)
- ✓ Hook up manual Draft creation

**Figure 6:** Unscoped



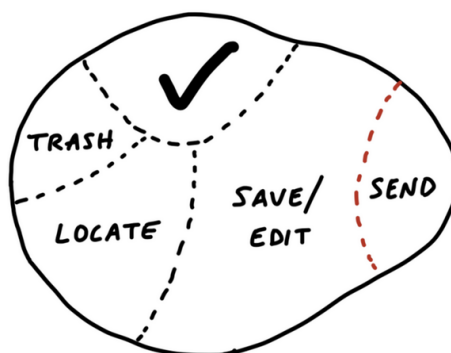
- ≡  **Start New**
- ≡ ☐ Design "Only you can see this draft" wrapper/labels
- ✓ Allow 'invalid' drafts to be created (without an addressee, for example)
- ✓ Hook up manual Draft creation
- ≡  **Unscoped**
- ≡ ☐ Intercept attempts to reply to Topic if a draft from a different message exists
- ≡ ☐ Handle draft message timestamps after sending
- ≡ ☐ Hook up Send from Draft edit state
- ≡ ☐ Remember draft content when editing draft
- ≡ ☐ Remember addresses when editing draft
- ≡ ☐ Hook up re-saving from Draft edit state
- ≡ ☐ Reduce duplication in Draft forms
- ≡ ☐ Design index of existing Drafts
- ≡ ☐ Hook up Draft deletion from Draft edit state

**Figure 7:** Scope one piece



- ≡ ☒ **Unscoped**
- ≡ ☐ Intercept attempts to reply to Topic if a draft from a different message exists
- ≡ ☐ Handle draft message timestamps after sending
- ≡ ☐ Hook up Send from Draft edit state
- ≡ ☐ Remember draft content when editing draft
- ≡ ☐ Remember addresses when editing draft
- ≡ ☐ Hook up re-saving from Draft edit state
- ≡ ☐ Reduce duplication in Draft forms
- ≡ ☐ Design index of existing Drafts
- ≡ ☐ Hook up Draft deletion from Draft edit state

**Figure 8:** Get one piece done



- ≡ ☒ **Send**
- ≡ ☐ Hook up Send from Draft edit state
- ≡ ☐ Handle draft message timestamps after sending
- ≡ ☒ **Save/Edit**
- ≡ ☐ Intercept attempts to reply to Topic if a draft from a different message exists
- ≡ ☐ Remember draft content when editing draft
- ≡ ☐ Remember addresses when editing draft
- ≡ ☐ Hook up re-saving from Draft edit state
- ≡ ☐ Reduce duplication in Draft forms
- ≡ ☒ **Trash**
- ≡ ☐ Hook up Draft deletion from Draft edit state
- ≡ ☐ Design a way to trash drafts from the index of drafts
- ≡ ☒ **Locate**
- ≡ ☐ Design index of existing Drafts
- ≡ ☐ Design a way to navigate to Drafts via "Inbox..." menu

**Figure 9:** Start splitting the rest

And you can end up resplitting as new tasks are discovered.

## Discovering Scopes

Scope mapping isn't planning. You need to walk the territory before you can draw the map. Not about tidiness, they reflect the real ground truth of what can be done independently.

## How to know if the Scopes are Right

Well-made scopes show the anatomy of the project.

Three signs indicate when the scopes are right:

1. You feel like you can see the whole project and nothing important that worries you is hidden down in the details.
2. Conversations about the project become more flowing because the scopes give you the right language.
3. When new tasks come up, you know where to put them. The scopes act like buckets that you can easily lob new tasks into.

On the other hand, these three signs indicate the scopes should be redrawn:

1. It's hard to say how "done" a scope is. This often happens when the tasks inside the scope are unrelated. If the problems inside the scope are unrelated, finishing one doesn't get you closer to finishing the other. It's good in this case to look for something you can factor out, like in the Drafts example.
2. The name isn't unique to the project, like "front-end" or "bugs." We call these "grab bags" and "junk drawers." This suggests you aren't integrating enough, so you'll never get to mark a scope "done" independent of the rest. For example, with bugs, it's better to file them under a specific scope so you can know whether, for example, "Send" is done or if you need to fix a couple bugs first before putting it out of mind.
3. It's too big to finish soon. If a scope gets too big, with too many tasks, it becomes like its own project with all the faults of a long master to-do list. Better to break it up into pieces that can be solved in less time, so there are victories along the way and boundaries between the problems to solve.

## Chowder

There are almost always a couple things that don't fit into a scope. We allow ourselves a "Chowder" list for loose tasks that don't fit anywhere. But we always keep a skeptical eye on it. If it gets longer than three to five items, something is fishy and there's probably a scope to be drawn somewhere.



## Mark nice-to-haves with ~

To be able to sort based on **must-haves** from the **nice-to-haves**.

## Chapter 13: Show Progress

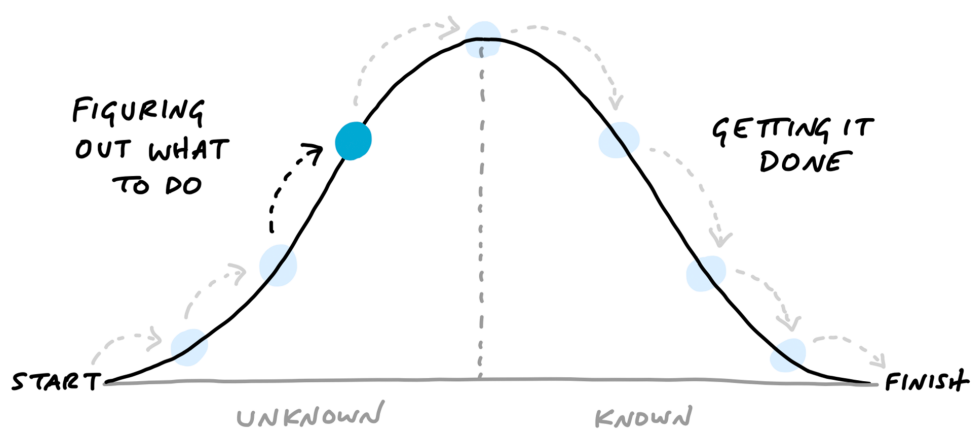
Asking for feedback feels like nagging.

The number of tasks grow with time since newer tasks are discovered.

## Estimates don't show Uncertainty

Some tasks have been done before so we know how long they'll take. Some are new tasks and can take 4 hours or 3 days. Putting that as a time estimate is a bit weird.

## Work is like a Hill

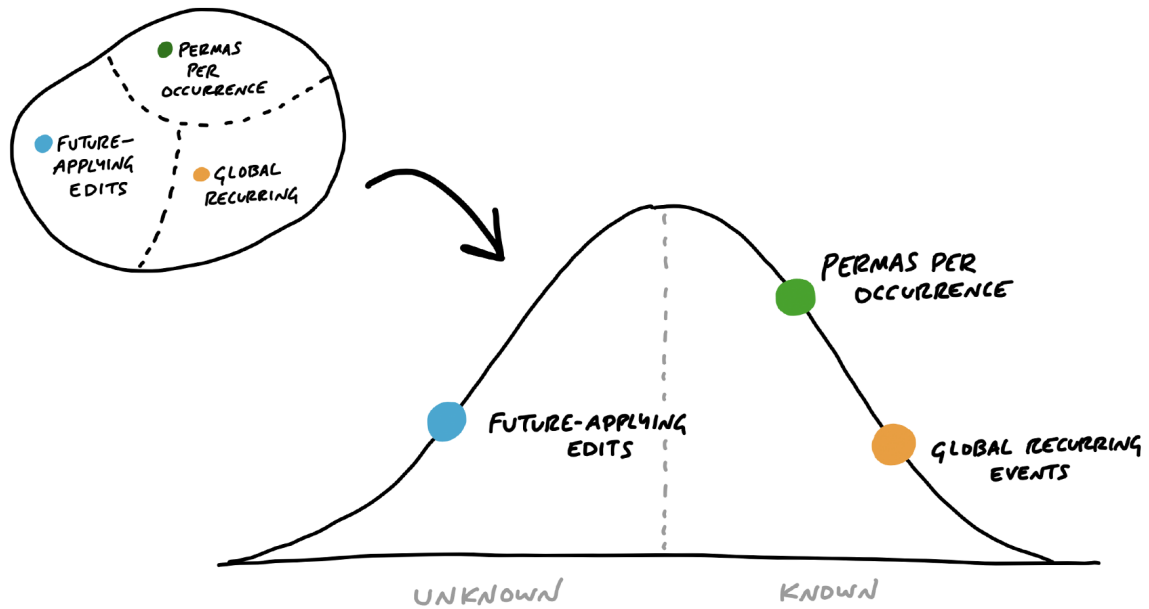


**Figure 10:** Unknown to known

Example goes through hosting a dinner party. At the beginning, you've only set a date. Next choice is to decide what kind of cuisine to do since some are vegetarian. You pick indian. Narrow on a dish. You reached the peak of the hill because from there, it's quite obvious what needs to be done: groceries, cook, host, clean.

## Scopes on the Hill

Use the same scopes as before but place on the hill. Is the scope going “uphill” or “downhill”?



**Figure 11:** The Hill



**Figure 12:** How things are moving on the hill

Manager doesn't need to ask the team where things are, can just look at the hill. If something hasn't moved, he can then ask "what's holding this back?"

### **Nobody says “I don’t know”**

The way they *say* it is by having the ball stuck in the same spot for a while. If this happens, maybe the person needs someone a little more senior to help out or to clarify.

Once it’s been spotted, the language of uphill/downhill facilitates the conversation. It’s less about the person (Looks like you’re stuck!) and more about the work. The question is: What can we solve to get that over the hill?

### **Build your way uphill**

Some teams struggle with backsliding when they first try the hill chart. They consider a scope solved, move it the top of the hill, and later have to slide it back when they uncover an unexpected unknown.

When this happens, it’s often because somebody did the uphill work with their head instead of their hands. Coming up with an approach in your head is just the first step uphill. We often have a theory of how we’ll solve something—“I’ll just use that API”—and then the reality turns out to be more complicated.

Do instead, by thirds: 1. I’ve thought about this 2. I’ve validated my approach 3. I’m far enough with what I’ve built that I don’t believe there are other unknowns.

### **Solve in the right Sequence**

Push the scariest work uphill first. “If you have to eat a frog, eat it in the morning. If you have to eat two frogs, eat the big one first.” Mark Twain quote interpretation.

Inverted pyramid like in newspapers: critical information first, following is less and less critical and more like “nice to haves” and “maybes”. You don’t lose context.

## **Chapter 14: Decide When to Stop**

At the end of the six week cycle, there should be a product coming out of it. Is this ready to release?

Focus on what the **baseline** was before the project, not on what the ideal best could’ve been.

### **Scope Hammering**

The scope is like grass, you don’t see it from the macro-level, but from the micro-level it keep growing. The team needs to be able to cut the scope.

Given the six week cycle to break the circuit, cutting is more akin to hammering. Making it fit inside the box. We ask ourselves:

- Is this a “must-have” for the new feature?
- Could we ship without this?
- What happens if we don’t do this?
- Is this a new problem or a pre-existing one that customers already live with?
- How likely is this case or condition to occur?
- When this case occurs, which customers see it? Is it core—used by everyone—or more of an edge case?
- What’s the actual impact of this case or condition in the event it does happen?
- When something doesn’t work well for a particular use case, how aligned is that use case with our intended audience?

### QA is for the Edges

At Basecamp, there’s only one QA person. They hunt for the edge cases around the core functionality.

The QA limits their attention to edge cases because the designers and programmers take responsibility for the basic quality of their work. They write their own tests and ensure that it works (they are responsible!).

QA == level-up != gate or check-point that all work must go through. Much better with QA but it’s not necessary.

QA generates *discovered tasks* that are *nice-to-haves* by default. The designer-programmer team triages them and, depending on severity and available time, elevates some of them to *must-haves*. Collect incoming QA issues on a separate to-do list, then if the team decides it’s a *must-have*, they drag it to the list for the relevant *scope* it affects.

Treat the code review the same way. Can ship without a code review. If time-permissible, then have a senior person review the code but much more as a teaching opportunity than creating a step in the process that must happen every time.

### When to Extend a Project

The only time to allow a small extension is if the remaining tasks are *must-haves* that withstood every attempt to *scope hammer*. These tasks must be all *downhill*. No unsolved problems or open questions. Any *uphill* work == hole in the concept. Unknowns are too risky to bet on.

Generally, if extensions always occur and are dealt with during the **cool-down**, then there is a problem when shaping and having too large an **appetite**.

## Chapter 15: Move On

### Let the Storm Pass

Show this to customers but resist knee-jerk reactions. “Yes, but what about this thing?”. Give it 2-3 days and let it pass. Remember why the work was done in the first place!

### Stay Debt Free

If you say “yes”, then you committed and took on debt. Say “no” because these requests are considered new and **raw** and will need to be shaped.

Remember, the thing you just shipped was a six-week **bet**. If this part of the product needs more time, then it requires a new bet. Let the requests or bugs that just came up compete with everything else at the next **betting table** to be sure they’re strategically important.

## Conclusion

Key concepts:

- Shaped versus unshaped work
- Setting appetites instead of estimates
- Designing at the right level of abstraction
- Concepting with breadboards and fat marker sketches
- Making bets with a capped downside (the circuit breaker) and honoring them with uninterrupted time
- Choosing the right cycle length (six weeks)
- A cool-down period between cycles
- Breaking projects apart into scopes
- Downhill versus uphill work and communicating about unknowns
- Scope hammering to separate must-haves from nice-to-haves