The Cabbage Tree Method

Open Source Collaborative Product Development

The Cabbage Tree Method

Open Source Collaborative Product Development

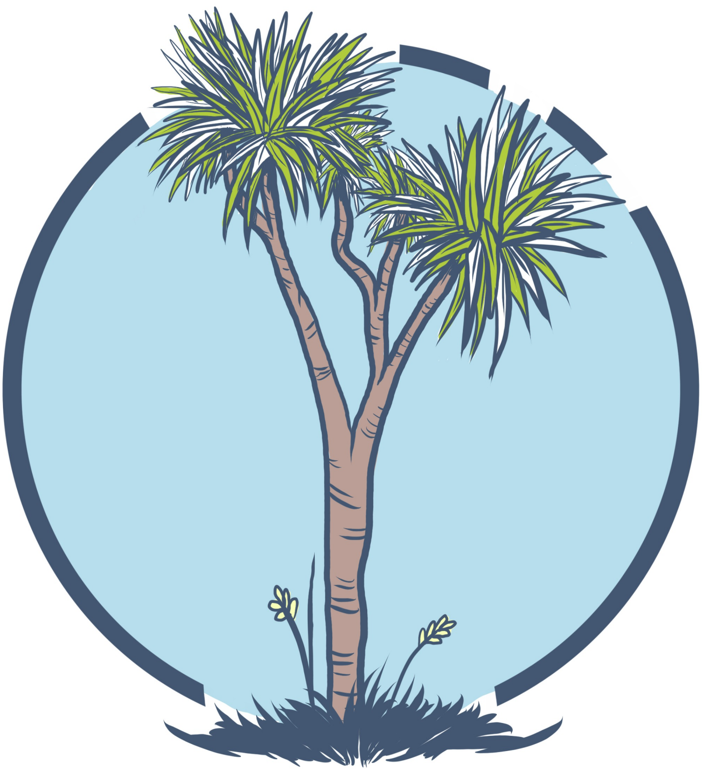
SHUTTLEWORTH FOUNDATION

Collaborative Knowledge Foundation

This book version is 0.1

Table of contents

Introduction



This book evolved from a need to choose or design a development methodology for the Collaborative Knowledge Foundation (Coko) which I co-founded with Kristen Ratan. Coko aims to reform scholarly publishing with modern HTML-first workflows, and we build open source tools to assist this goal. There are many development methodologies available, but I have found many of them unsatisfying. What I was missing was something that articulates many of the principles of the Agile Manifesto and Kent Beck's Extreme Programming methodology but applicable to open source culture. Essentially these two, and other, forces at play within the history of Systems Develop Life Cycles, branch away from the top-down design and development methods that preceded them.

However, while I see these kinds of ideas as potentially transformative, I haven't seen them well executed, nor does it seem to me that they have really changed open source processes all that much.

On reflection, open source itself seems to be a "culture method" all of its own. That culture is based largely on the idea of developers having their own itch to scratch, as expounded by Eric Raymond, and sits on its own branch of the SDLC tree. While open source is, in a sense, a radical proposition in itself, open source also seems oddly unaffected by some of interesting changes occurring in the proprietary software world.

At the same time, and from a completely different sector, I have been inspired by John Abele. For me, this is an unusual source of inspiration. John focuses on the development of proprietary hardware tools for the medical sector. Yet he's a generous mind and has written a lot about the collaborative design of surgical instruments in the 1960s and how collaboration itself, involving the end user, is the key to overcoming a hostile market. John and his colleagues successfully championed the introduction of non-invasive surgical techniques at a time when surgery *meant* cutting. His ideas fascinate me and I mined the discussions we had for clues about how open source could leverage the techniques John expounded.

On further reflection, all this searching was, in a way, a little ridiculous since it led me full circle, back to my roots. I realised that the two critical ingredients required to enable the production of market-beating open source products were 1) the involvement of the people who needed the product in the design process, and 2) the effective facilitation of the design process. It was full circle because this is exactly what I'd been doing for the previous eight years — but with books, not software. From 2007, I had been trying to work out how to produce books quickly (initially for the production of free manuals about free software) and through many dead-end, frustrating explorations over a period of four years, I developed and refined the Book Sprints Methodology. With Book Sprints, the idea is to facilitate the people who need a book to collaboratively write it themselves. When

I understood this framing of Book Sprints and how it tied into both John's ideas and software development, everything started to fall into place.

The next step for me was to work out how this approach could be applied sensitively within open source culture, or more specifically, within the Collaborative Knowledge Foundation. That was largely a process of mapping what I knew from Book Sprints to what I knew of software development. It's strange that the solution for my problem resided in allowing two parts of my brain to cross pollinate — opening up previously firewalled learnings and letting them talk to each other. Since I knew from experience the dynamics of creating a method, I wrote out the method rather quickly at this point. I already knew that methodologies result from making assumptions and then holding your breath and trying them out in the real world. So that's what I did. Thankfully, I had others who held their breath with me, most notably and thankfully Kristen Ratan to whom I'll be forever grateful for her insights and trust and patience with me, and Karien and Helen from the Shuttleworth Foundation, and all of the generous Shuttleworth Fellows, who listened to me bang out the rudimentary ideas and offered insightful critiques and expansions.

I've now tested out the process and I'm extremely happy with how it works. It's still in need of more real world scenarios, as more testing strengthens any methodology. And that's why you are reading this now. I hope you'ill read this book, think about it, and try out the method I describe on these pages. The more people who try this, the faster we can learn from each other. If you do choose this path, I hope that together we'll transform open source and make it into the world-beating producer of user facing products that it should be.

adam@coko.foundation

How this Book Was Written

I fleshed out the concept of the Cabbage Tree Method over a number of weeks and then months as I tried it out, posting to my blog as I went along. During that time, I received valuable feedback and insights from Kristen Ratan and many of my fellow Fellows at the Shuttleworth Foundation as well as the Foundation's staff, mainly

Arthur Attwell, Seamus Kraft, Steve Song, Andrew Rens, Ryan George, Helen Turvey, and Karien Bezuidenhout. Also, as always, I got amazing feedback, support, and mentorship from Allen Gunn (Gunner). It is worth noting that Gunner has developed similar methodologies and his organisation — Aspiration — is an invaluable source for information, experience, and wisdom on these topics.

I also learned a lot from the Coko team who were participants in the development of products using this method, including the talented trio of Yannis Barlas, Christos Koksias, and Julien Taquet who bore the brunt of it, as well as Alex Theg, Jure Triglav, Charlie Ablett, and Wendell Piez. Others also provided great feedback including Micz Flor, Eleni Michaelidi, and Nicole Martinelli.

The good people at the University of California Press and California Digital Library were also the first case use specialists to go through a full Cabbage Tree Method cycle. Many thanks to Erich van Rijn, Catherine Mitchell and Justin Gonder, and especially thank you to Kate Warne and Cindy Fulton for bringing use case expertise and generous spirits to the table.

I then compiled some of these posts into this book and asked Scott Nesbitt and Pepper Curry (illustrations) to help improve it. Raewyn Whyte then cleaned it up, making it proper grammar, and Julien Taquet made a book out of it!

I highly recommend the skills of Pepper, Raewyn, Scott, and Julien if you wish to produce a book of your own.

Finally, I am so very grateful to the Shuttleworth Foundation for choosing me as a fellow. Without their support, Coko, this method, and this book, would not have been possible. Due to the open and supportive environment the Shuttleworth Foundation works hard to create I have had the freedom to express my own views. It follows that the views expressed in this book do not necessarily reflect those of the Shuttleworth Foundation.

Thoughts About Open Source

Open source is broken. More precisely, it's partially broken. One part of the open source development model works very well, but the other part is failing users.

The part that works is where the famous 'itch to scratch' model comes into play. If a developer has a problem that they can fix with some new code, then they're in a good position to do just that. However, while writing code is a necessary condition for creating software, it's not the critical reason this model works. The critical reason is that, in these cases, the developer is the user. The developer understands the problem in depth because it's their problem.

But outside of developers solving their own problems, open source has largely failed users. There are few user-facing open source solutions that can beat their proprietary rivals in terms of approach, utility, and usability. I count Unity, GitLab, and Mattermost amongst those few. But there should be many more. Why is it this way? It's because we've incorrectly concluded that the ability to develop software is the same as the ability to solve any problem that involves software.

Understanding the problem, developing an approach to a solution, and developing software, are three very different skills. Open source culture has not, by and large, recognised the differences between those three skills. Instead, we've conflated those skills into one: *the ability to develop software*.

We need to move beyond that way of thinking.

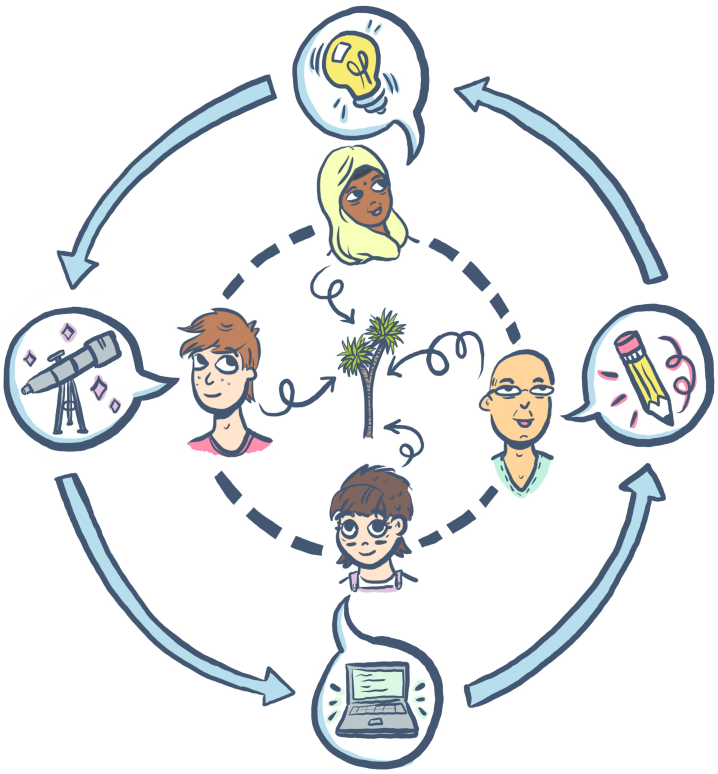
By focusing too heavily on developing software, we’ve forgotten that "developing software" isn’t what we’re doing. We're actually trying to solve a problem for someone. And that someone is often not a developer. That someone is usually the person who will use the software; and software, open or closed, is useless if it doesn't address that person’s problem. Hence the primary goal of open source is to solve problems, not create code, even though software is the dominant means to get there.

That key idea has gotten lost over the years. It’s led to a very developer-centric culture that sees all problems as issues developers haven’t yet tackled. It’s led to an over-reliance on technical thinking. It’s led to a lack of cultural diversity that doesn't reflect the backgrounds of the people using the software, and which can lead to poor assumptions and bad solutions. It’s led to a lack of diversity in the roles within open source projects and the power imbalance that accompanies this. It’s led to a lack of understanding of the wider issues of empowerment. Most of all, it’s led to the marginalisation of skilled people who are not developers and to the marginalisation of the people whose problems open source is trying to solve.

We need to change this situation, and we need to change it now.

What can we do? The answer is simple: **always have the people with the problem at the heart of an open source project**. We need to remember that the real strength of open source is the insights that the user brings to a problem, insights that no one else has.

While the answer is simple, its implications are huge. Those implications include diversifying participation, making users central to the project, tearing down technical meritocracy as the single determinant of value, and experimenting with new models of open source culture.



This is why I developed the Cabbage Tree Method. The Cabbage Tree Method advocates, by example, for a fresh approach to how open source projects are created, constituted, and run. I very much believe that the Cabbage Tree Method isn't just a design methodology: it's a template for a different model of organising open source projects. A model that, at its core, requires and promotes a more inclusive culture.

What Is the Cabbage Tree Method?

Imagine a bunch of people in a room, all sitting around a table. There are some whiteboards in the room, and coffee, sticky notes, and maybe even a data projector, litter a table. All of those people, except one, share a common problem and they want to create new software to solve it.

But where do they start? There are no developers here ... what's going on? One of them, the facilitator, steps up and initiates a short period of introductions and then asks the question "What is the problem?"

From this, a process unfolds where the people who need this new software (let's call them the *use case specialist*s) explain all their frustrations with the ways things are done now and what could be better. It is a wide-ranging discussion and everyone is involved. At the facilitator's prompting, someone jumps up and draws a straggly diagram of a workflow on one of the whiteboards to get their point across. Another pipes up to add nuance to one part of the diagram because they fear the point wasn't adequately understood. There are some quiet moments, some discussion, lots of laughter, a break for lunch. Plenty of coffee.

Throughout the day, the group somehow (the facilitator knows exactly how) evolves their discussion from big picture problems and ideas to a moment where they are ready to start designing some solution proposals. The facilitator breaks them into small groups and each group has 45 minutes to come up with a solution. When they come back, each group presents their ideas. Some of the ideas are very conceptual, almost poetic. Other ideas are very concrete and diagrammatic. Everyone thinks carefully about the merits of each proposal and what it is trying to say. Discussion ensues. Members of the group ask clarifying questions. After all the proposals are made, they decide on an approach.

In a short time, they have agreed on a set of requirements for software that they have consensus on and all believe will solve (at least some of) their problems. They take photos of all the whiteboard diagrams and document the design agreements thoroughly, creating a Design Brief. At the end of the day, they walk out the door and the Design Session is over.

The next day the build team, featuring user interface (UI), user experience (UX), and code specialists, looks over the documentation with the facilitator through remote conferencing. They discuss the brief, what is clearly defined and what is still to be defined. They work through the issues together, jamming out approaches to open-ended questions which are both technical and feature-focused. The session is not long, perhaps two hours. It's a lot of fun. From this session, the Design Brief is updated with the decisions. Many technical solutions are left wide open for the code specialists to think through and solve over the next weeks. However, the code specialists can, and do, start work immediately, though the UX specialists add mockups to the documents over the next days. The team works things out on the fly where necessary and gets onto it. Over the next weeks, a few questions to the use case specialists surface — these are either asked directly or through the facilitator.

The use case specialists reconvene six weeks later with the facilitator and are presented with the working code that has been created by the build team over that period. Everyone is amazed. It's just as they imagined, only better! After seeing the working code, they each have further, exciting, insights into how this problem might be solved. The facilitator steps up and they go through it all again to design the next part of the solution. Everyone is bursting to have their say.

The design-build cycle is repeated until they are done and the software is in production.

This is the Cabbage Tree Method.

The Cabbage Tree Method (CTM, for short) is a new way to create open source software products. With CTM, the people who will use the software drive its design and development under the guidance of a facilitator. It's a strongly-facilitated method that generates and requires immersive collaboration.

You can think of CTM as a new branch on the Systems Development Life Cycle (SDLC for short) tree. Some popular SDLC methods include Spiral, Joint Application Design, Xtreme Programming, and Scrum (some of which conform to the values of the Agile Manifesto — see http://agilemanifesto.org/ for details).

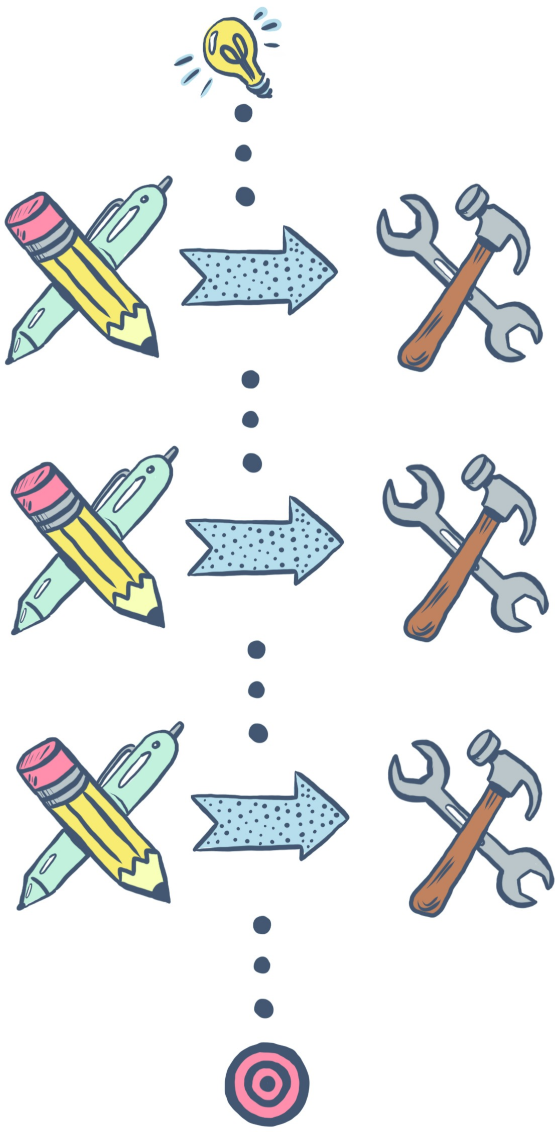
Unlike the various SDLC methods, CTM is specifically aimed at the free software/open source sector. In that sector, the cultural rules are quite different from environments where teams are employed to work within a more formal business or corporate structure. However, CTM differs from open source processes that have embraced developer-centric solution models, thanks to its focus on users designing the software with a facilitator as an enabling agent.

What also sets CTM apart from other methods of developing software, is that it doesn't have:

- personas
- avatars
- user validations
- user stories
- empathy boards (etc)
- 'experts' designing the solution for the user

This process isn't about development procedures that represent the user at a distance. It's about communicating and collaborating with the user at the centre of the process. It's not a question of profiling a so-called user, or turning them into an avatar or proposition, or trying to generate empathy with them from afar. Rather, a core requirement of CTM is to directly involve in the design process everyone who will use the system. The idea is that if you want to know what the user wants, don't imagine their response. Ask them.

The Cycles of CTM



Like most modern SDLC methods, CTM is iterative and has clear cycles. Each cycle consists of a Design Session followed immediately by a Build Period. These cycles repeat (design, build, design, build, design, build etc) until the solution is complete.

Design Sessions

The specialists most in demand for the Design Sessions are the use case specialists — the users themselves. The Design Sessions are always conducted in person — they don't work well with remote participation. Each Design Session can be as short as two hours, or as long as one day.

The general principle of the Design Sessions is that all users affected by the software must be present at the appropriate moment – either in total or as a representative group. Without their presence, a solution cannot be developed. A fundamental rule of CTM is that no one speaks for the users other than the users themselves.

From each Design Session, a short brief is created that describes what has been agreed to, what is absolutely required to be done in the following build period, and what is left to be solved during the build period.

Build Period

The build period takes place immediately after the Design Sessions. The build period can occur remotely and may take two to eight weeks, perhaps longer. Building is the job of the UI/UX and code specialists and it is here they can both be creative and exercise their User Interface (UI for short) / User Experience (UX) and programming skills. Use case specialists don't participate in the build period but may be consulted for clarification during this period.

Before the build period begins, each of the build team members receives for consideration the initial brief that was created during the Design Session. The build period then begins with a meeting where the code and UI/UX specialists discuss the brief, decide on an approach, and together develop solutions for any outstanding issues. This may include solving some complex feature, technical, and usability problems — essentially working out how to achieve everything the users have already decided, plus designing what is left over.

Then briefs are written and agreed upon, mocks done where necessary, and building begins.

Getting Started

The Set Up

Before you can start using the Cabbage Tree Method to develop your solution, you'll need:

- a problem that needs to be solved
- a facilitator
- stakeholder commitment
- a venue for the Design Sessions
- a goal
- user interface/user experience and code specialists

The Problem

The people with the problem (the use case specialists) should voice the problem. You can't, and shouldn't, imagine problems for them. The initial articulation of the problem space is really the starting point for the process. The facilitator will need to work with the group to get a more accurate understanding of the problem at hand. It should be as detailed as possible and clearly documented.

The Facilitator

Central to this process is the facilitator. There are many models for facilitation and still more that call themselves facilitation. The facilitator is not the Benevolent Dictator For Life, a cat herder, nor a Community Manager as commonly found in the open source world. The CTM model facilitator is closer to an unconference-style facilitator, expert in managing power dynamics on the fly and enabling people to converse and work together in a very collaborative and egalitarian way.

However, a CTM facilitator is not an unconference facilitator. A CTM facilitator is someone who uses unconference tools to manage dynamics, but who must also be able to drive people through the CTM method to specific, clear end points. These two aspects of the job don't always come together. Don't assume that an unconference facilitator can perform this role.

Stakeholder Commitment

You need the commitment from the people who will use the system — these are your primary stakeholders, your use case specialists. They'll design the software, so its success will be a direct result of their participation. If an important stakeholder is missing, then you can't design for them. You need to make sure you have everyone affected by the system present and that they're committed to this process.

A Venue

A good venue, at a minimum, is a space with just enough room for a table and enough chairs to seat everyone. A long whiteboard wall is good to have but you can substitute large pieces of paper (like flip charts) if no whiteboards are available. The ideal venue would have breakout spaces and fresh air. If you're working with a single organisation, then it's good to have the sessions in rooms connected to, or a part of, this organisation's workspace. This enables them to demonstrate in-house legacy tools or their current workflow, if necessary. Coffee and food are always important to have at hand.

A Goal

The stakeholders need to articulate a clear goal. That goal could be as straightforward as stating *We need to replace our existing system with X*. It doesn't need to be much more detailed than that. You can determine the scope of X in the first Design Session.

User Interface and Code Specialists

While you could try developing software without UI/UX and code specialists — I've done it many times when I had no money to hire specialists — you won't end up with a good product. UI/UX and code specialists add a lot of value when you reach CTM's Build Period.

Design

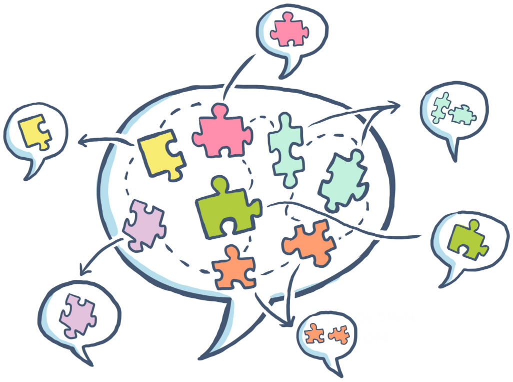
The Design Session

All Design Sessions follow the same basic pattern, but the first session needs to cover a lot of ground that is not repeated in subsequent sessions.

The basic structure for every Design Session is :

- where are we now? (review where the group is now)
- where are we going? (consider options on what a solution might look like)
- how do we get there? (design the solution)

During the first Design Session, the group needs to spend a lot of time detailing what the current issues are, how they think those issues could be solved, and evolving a final high-level approach to resolving those issues. Subsequent Design Sessions then break off a part of this higher level approach and design a solution for each piece of the problem, one piece at a time.



Since the first Design Session is, as we say in New Zealand, 'the same but different,' I've outlined that below in detail and followed with short notes on how the subsequent sessions differ from the first one.

A full day may very well be needed for the first Design Session; even need two days might be needed. The subsequent Design Sessions may need only a few hours each. Each session should comprise between 2 and 12 people. There should be as much continuity of participants over all sessions as possible.

What follows is more of a guide than a set of rules. As you'll discover, facilitation of the Design Sessions is more art than science. The facilitator must be responsive to the group's needs and adjust the process accordingly. Think about what is happening and don't blindly follow the method.

The First Design Session

In the very first Design Session, a lot of background needs to be covered. You should spend time understanding the 'big picture,' refining it to a manageable problem to solve, and finally, work out what the solution needs to look like.

Phase One – Why Are We Here?

This phase is most important in the first Design Session. The key question may need to be revisited periodically should the mission change or the group bring in new members in subsequent sessions.

This phase consists of:

Introductions — Even if everyone knows each other, always do a round of introductions. It sets a baseline expectation that you should state even the obvious. It will also help you to understand something about the stakeholders present and the roles they play. Experienced facilitators also use this time to read the interactions in the group and start formulating strategies to get them collaborating.

Asking why we are here — Any new software brings about change, change in how people do things, change in what kinds of things exist in the world. So, it is important to ask the question *Do we want change?* This should be explicit, and the facilitator can actually ask this question outright. The answer is *yes*, but it's not the answer that it is important. What's important is the affirmation by the group that they are prepared to undergo a process that will change the way things are currently done. This also means they need to make a commitment to using the software developed through this process and you should ask them for this commitment also.

Phase Two – Where Are We Now?

Phase two is a review. This review will be very detailed as a very deep, shared understanding of the problem at hand needs to be achieved. The review consists of:

Asking where we are now — Ask the group what the problem is that they are trying to solve. A good, shared understanding of the 'big picture' problem you are all trying to solve together is needed before it can be filled in with concrete detail. For example, if you are working with an organisation to fix workflow issues, this will require a thorough documentation of the current workflow. Start with a description of the current problem, identifying current processes and pain points in as much detail as possible. This will take a lot of discussion and it may be necessary to ask participants to draw or demonstrate the issue they are talking about. Document it all thoroughly.

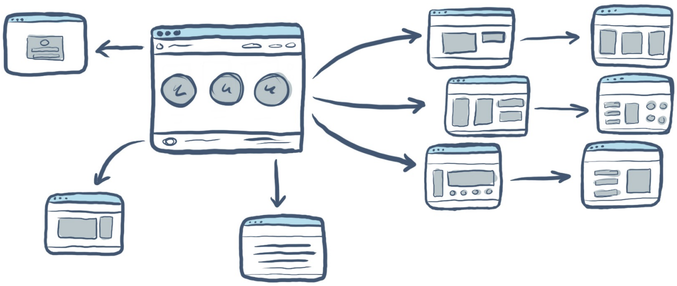
Identifying the scope — The problem you identify above may be huge so you will need to spend time narrowing down the problem space with the group. This might mean asking many clarifying questions and teasing out logical inconsistencies to get a very clear, concrete, understanding of the problem to be addressed. It might very well be that you discover the problem resides elsewhere than originally thought, or it could be that you segment the problem and choose to tackle just one part of it. Whatever the outcome, make sure everyone explicitly agrees to it.

Phase Three – Where Are We Going?

During this phase, the group articulates what the solution will look like. This can be a free-ranging, dreamy discussion but should be shaped slowly (like whittling wood) into something reasonable and achievable. You're looking for 'big picture' approaches to the problem, so spend as much time as necessary pitching ideas around and exploring possibilities. Funnel these ideas through discussion until you have a general consensus on a *very* general approach to the problem. Don't worry if this feels very abstract and 'up in the air'. This is normal. The next step is when you refine these ideas to make them concrete. Also, don't be surprised if this process alters the understanding of the scope of the problem. Feel free to allow phases two and three to feed back into each other. You will need, however, to keep the discussion moving forward and be prepared to break people out of any cyclic problem holes they might get stuck in.

Phase Four — How Do We Get There?

In the first Design Session, you start the process for what I call a Solution Proposition. The Solution Proposition is a design for the 'big picture' approach you will take to solve the problem. For example, if the solution is a web platform, the Solution Proposition might look like a drawing of all the relevant pages of the platform and what they do. A good target outcome of this process is to produce high-level wireframes of the solution. Although the Solution Proposition is at a very high level, it will require a lot of "on the ground" detail, thinking, and discussion.



There are a number of ways you can develop the Solution Proposition. The method you choose is a product of your observations about how the team works together, the problems they are trying to solve, and your experience as a facilitator.

Below are some example processes to help get you there. Remember, however, that facilitation is about *invention*. Feel free to invent processes like the ones below, on the fly. Test them out in your sessions and learn. If you are doing your job well, then you'll be experimenting all the time. If you are doing your job *really well,* then no one will notice when one of your new inventions (possibly made up as you're saying it) fails.

The ones listed below work, but the first time I used each I made them up on the spot (but let's keep that between the two of us!):

1 **Blank Canvas**
Start with a very open-ended session. Ask how the group would like to work and let the conversation roam. Document points that seem important. Document repeated themes. What you're looking for is a starting point for the new story about how things will be done. This starting point might be very concrete, or it might be highly conceptual. It could also be that you think you have the starting point but find that some people in the group want to take the starting point back to the fundamentals of what they are trying to achieve. Your job as a facilitator is to witness this and keep the process going until you find the point where most, if not all, of the participants, want to start the new story.

At this point, you need to make sure the new story is well documented and you're getting good clear, simple, points. At this point, there's no substitute for having experience as a facilitator. If you're struggling, then the best strategy is to try and make the starting point somewhat concrete and less conceptual. For example, draw a blank box and say something like *How do we start the new process in a browser?* Then ask participants to draw out concrete details about what *could* happen in this new canvas. Let people roam and use what you've learned in the previous phases and what you know about the domain, human behavior, and technology to keep the conversation real. This will make your work much easier.

Iterate like this and work towards building concrete steps that illustrate the new ways of doing things. It's surprising that in some situations, this process will lead directly to a solid understanding of the user interfaces and flow you are building. At other times, you may need to keep the conversation going until something concrete materialises. In general, the more experienced you are as a facilitator, the more smoothly this part of the process will proceed.

2 **Pitching**
Give each of the participants (or small groups) large pieces of paper and ask them to draw a solution. Set a time limit for this task. Don't let anyone claim they *don't know enough* about the problem space — seemingly naive Solution Propositions often have very insightful points. At the end of the time limit, each participant will pin their paper to the wall and speak about it for five minutes. These are not real pitches, just a quick presentation that explains what they were thinking. Then, have five minutes of questions and comments immediately after each pitch. Leave all papers on the wall — just add to them for every pitch.

After everyone has presented, you should break for a while as this was probably a long session. It's also good to break here to take some time and consider what your next step will be. You must derive a single Solution Proposition from the pitches. It's OK at this point to propose it yourself when the group returns to the room. Make sure that when you do this, you're proposing ideas that the group has had, not your own pre-designed solution. Invite comment on the Solution Proposition that you propose.

This phase should end with everybody having a good understanding of what cultural changes are necessary and what technology changes (what you're trying to build) are required. It's not unusual to have whiteboard wireframes at the end of this process.

3 **Give Them the Pen**
Sometimes it's very useful to give someone with a strong vision for the solution a pen and an opportunity to draw the solution on a whiteboard and explain it. This turns the idea into something tangible that the group can then discuss.

Phase Five — Summarise and Capture

This phase involves wrapping up the session, summarising the agreed problem space, the Solution Proposition and, most importantly, what will happen next. Document all this thoroughly and ensure everyone agrees to it.

These summaries will be distilled into a Design Brief that will then be passed onto the build team (see following section).

Phase Six – Working Agreements

During this phase, you'll agree upon:

- how you'll work together
- what channels you'll use for communication
- where you'll store all the agreements, supporting documents, and the like
- when the next Design Session will take place

Build

Now you come to the build period. See the chapter The Build Period for information on how to effectively facilitate the build period.

Subsequent Design Sessions

In the subsequent Design Sessions, you'll be guided by the basic Solution Proposition that the group has agreed upon and there will be some software (created during the Build Periods) to review.

The subsequent Design Sessions follow the same basic pattern and rules of the first Design Session, with one exception: instead of focusing on the problem as a whole, the group focuses on developing solutions to specific portions of the problem.

The Design Sessions move incrementally forward in this fashion: what has been built is reviewed, the next issue to be solved is identified, and the next part of the Solution Proposition is designed. This enables the Cabbage Tree Method to advance step-by-step, filling in details of the vision and building the necessary features in turn until the entire Solution Proposition is realised.

As with the initial Design Session, you need to keep asking the following questions in each subsequent one:

Where Are We Now?

This question isn't an audit of the problem you're trying to solve. Instead, use it as the starting point to review any work from the previous build period. Discuss it in detail and compare it to the previously agreed-upon designs. If there are changes to be made, work through them now and document them. You can pass these changes to the build team at the end of the session as feedback and requests for changes. This step is also a good moment to add detail to the Solution Proposition or bring in any ideas that might improve it.

Where Are We Going?

Agreeing on the scope means deciding what part of the Solution Proposition you're going to work on next. Ask the group to define the most important problems to solve next, and, if possible, have the group choose a single, cohesive, problem to solve. Avoid trying to solve too much.

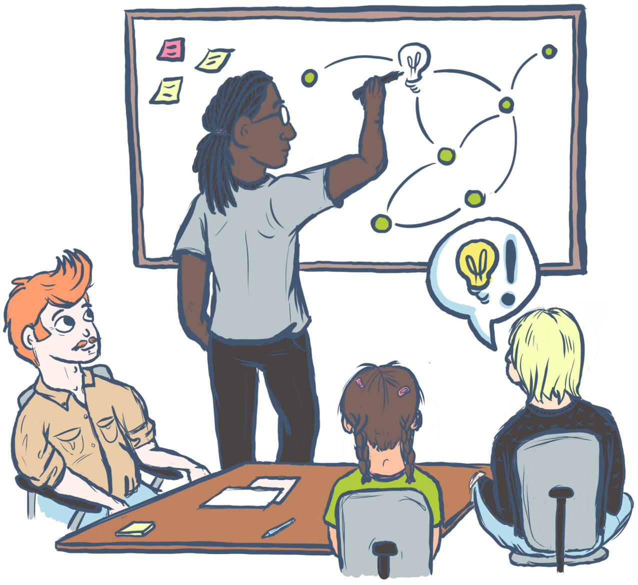
How Do We Get There?

Asking this question focuses the group on developing a design for the portion of the problem they've chosen to solve. You can also use the patterns from the first Design Session here to focus on what the group wants to tackle.

Summarise and Agree

As with the first Design Session, make sure you document everything and get explicit agreements on the details of the final design.

Effectively Facilitating the Design Sessions



Facilitation is a key ingredient for the Design Sessions. In fact, no group should attempt to use the Cabbage Tree Method or run a Design Session without a facilitator.

The facilitator is part of your open source team and should not be part of (if possible) the user group that needs the solution. This neutrality frees the facilitator, as best as possible, from existing ideas, and organisational politics and dynamics. In turn, this enables the facilitator to read the group clearly and interact freely.

Your tools for facilitating the Design Session are post-it notes, a large whiteboard and markers (or large pieces of paper).

Facilitation Tips

Facilitation is both an art and a science. It's difficult to master both aspects of facilitation. If you're new to facilitation, then it's best, if possible, to watch experienced facilitators in action and try some small experiments first. I strongly advise against learning facilitation from a book. There's no substitute for experience.

Having said that, here are some things to remember when facilitating Design Sessions. They're intended to help you formulate an approach to this process. They're not intended as a facilitation 'how to'.

Build trust before products — As a facilitator, your job is to build trust in you, the process, the group, and the outcomes. Each time you make a move that diminishes trust, you're taking away from the process and reducing the likelihood of success. Building trust, working with what people say and towards what people want, must be put ahead of building the product. Trust that the product, and a commitment to it, will emerge from this process, and the product will be better for it. I am thankful to Allen Gunn for this wisdom shared very kindly with me many years ago.

Change is cultural, not technical — Too often, people propose technology as the mechanism for change. Technology doesn't create change. People create change, with the assistance of technology. Make sure the group is not seeing technology as a magic wand. The group must be committed to changing what they do. The *what they do* shouldn't be posited as a function of technology. It's their behavior that will always need to change, regardless of whether there is a change in the technology they use. Behavioral and cultural change is a necessity, not something that *might happen*. The group needs to recognise this and be committed to changing how they work. This also means that they must commit to using the solutions they design together.

Even the best solutions have problems — Understand and embrace that even the best solutions, including the one you're representing, will have problems. There is no perfect solution. If there is, then, inevitably, time will make it imperfect.

Leave your know-it-all at the door — If you're part of an open source team, then you may be expected to be somewhat of a domain expert. It's not possible to leave that at the door, of course, but you should be careful that you don't use this knowledge as heavy handed doctrine. Instead, your domain knowledge should manifest itself in signposts, salient points, and inspiring examples at the right moments. These points should add to the conversation. They shouldn't override the conversation. Don't overplay your domain knowledge and vision. You'll lose people if you do.

Give up your solution before you enter the room — If you're part of an open source team, it's also a mistake to enter into a Design Session and advocate or direct the group towards your teams technology. If the group chooses a technology or path you don't have a stake in, then that's fine. The likelihood is that they'll choose *your* technology or approach, otherwise you wouldn't be asked to be in the same room with the group. However, if you facilitate the process and drive them to your project you'll be reading the group wrong and they'll feel coerced and, similar to above, you'll lose them.

Move one step at a time — Move through each step slowly. The time it takes to move through a step is the time it takes to move through a step. There is no sense in hurrying the process, as doing that will not lead to better results or faster agreements. It will, in all likelihood, move towards shallow agreements that don't stick and ill-thought-out solutions that don't properly address the problem. The time it takes to move through a step will also give you a good indication of the time you'll need for the steps to come. Be prepared to adjust your timelines if necessary, and to return to earlier unresolved issues if need be.

Ask many questions, get clarifying answers, ask dumb questions, the dumber the better — Try not to ask leading questions. Keep asking questions until clear, simple answers are given. Break down compound answers, where necessary, into fragments, and drill down until the necessary clarity is found. Dumb questions are often the most valuable questions. Asking a dumb question often unpacks important issues or reveals hidden and unchallenged assumptions. It's very surprising how fundamental some of these assumptions can be, so be brave on this point.

Reduce, reduce, reduce — The clearest points are simple ones.

Get agreements as you go — Double check that everyone is on the same page as the process proceeds. Get explicit agreement, even on seemingly obvious points. Total consensus is not always necessary or possible, but general agreement is.

Document it all — Get it down in simple terms on whiteboards or using whatever tools are available. At the end of sessions, document what has been learned with digital, shareable media.

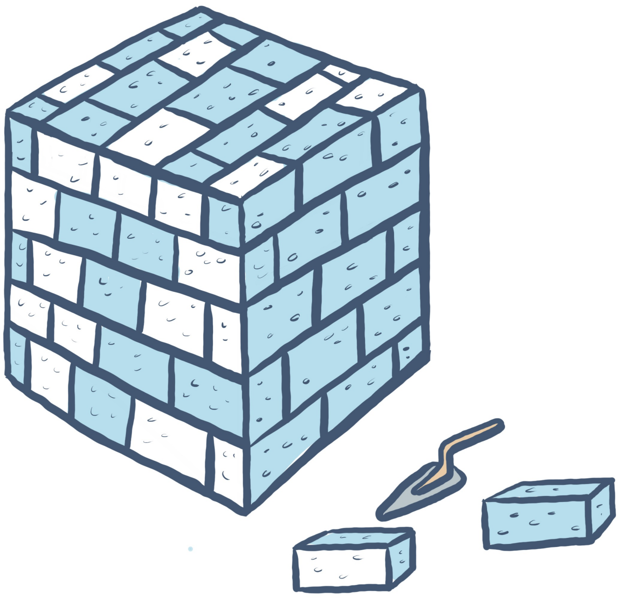
Summarise your journey and where you are now — At the end of a session, it's always necessary to summarise in clear terms the journey the group has taken and the point arrived at. Get consensus on this. It's also useful at various points throughout the session to do this as a way of 1) illustrating you're all on the same journey, and 2) reminding people what the session is all about. In the summary, make sure to illustrate that *you* understand the key points. Bring out points that may have taken a while to get clarity on or that you may have initially misunderstood. There will be many of those points if you're doing your job well! Doing that lets the group know you're embedded with them and not merely guiding them towards a predesigned end game.

Use their semantics — Use the group's terminology to describe the problem and the solution. Don't impose your own semantics. Remember, they are the experts. Use their language. If, instead, you use your terms in the process, then inevitably people will become confused. If a term doesn't exist for something, ask them to define it.

Remember, facilitation is an art of invention — At the very least, facilitation is an act of translating known patterns into a new context and tweaking those patterns on the fly. But the reality is that much of the what a facilitator does will involve making things up. Instinct and the fear of failure force the on-the-fly invention of methods, but a good facilitator makes it appear that the method has existed for a hundred years and has never been known to fail. When it does fail, a *really good* facilitator will ensure that no one notices and that the outcomes were the ones desired.

Build

The Build Period



The Build Period, as its name suggests, is the time when the code and user interface specialists start building. The Build Period follows immediately after a Design Session.

The team can choose its own build methodology. Some organisations are set up to follow what they call "agile" methods (I use inverted commas here for good reason — please see the critique of the Agile industry written by Pragmatic Dave, one of the developers of the Agile Manifesto, here: *https://pragdave.me/blog/2014/03/04/time-to-kill-agile/* or *https://www.youtube.com/watch?v=a-BOSpxYJ9M*). Or, they follow a process that demands more detail up front. The choice of build methodology doesn't matter as long as it is sensitive and responsive to the design process.

The build process discussed below, which I feel is "native" to CTM, has the following advantages:

- it's a great the opportunity for the UI/UX and code specialists to be creative,
- it's extremely fast and efficient,
- it delivers great results, **and**
- it actually works.

The Design Brief

The Build Period revolves around a single source of truth: the design brief. The brief comes out of the preceding Design Session. As you continue to work out details you must maintain the design brief with accurate and up-to-date information on the team's approach to the solution at all times. This should be kept somewhere so anyone can find it at anytime.

A first version of a high-level brief should emerge from each Design Session, or the next day at the latest, and then be sent to the UI/UX and code specialists. This brief doesn't need to be more than one to two pages long, and it should:

- articulate a clear goal for the Build Period
- explain the terminology (if necessary) to avoid confusion
- outline the design decisions; and
- identify what is still left to be decided.

The last point is very important. The brief should clearly state what design choices have already been made and what is left to be decided by the code and UI/UX specialists. This defines the 'scope' within which the build team can be creative (sometimes there are a lot of problems to be creatively solved, sometimes less so).

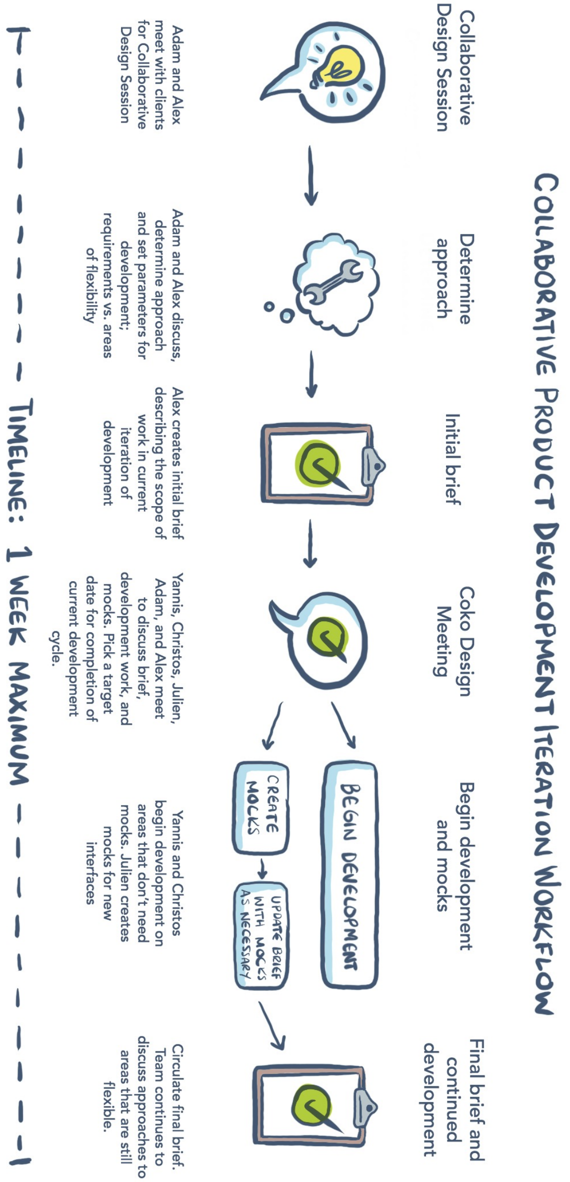
This document should be easy to read. It's largely a narrative document, not a requirements document.

The Build Meeting

The build team then reads the design brief and meets a few days later. When the team meets (if you can work out a good online whiteboarding system – BigBlueButton is a great choice — remotely works well), the facilitator leads them through a shorter design process. In this meeting they together consider the brief and discuss matters, working through issues as they go. The goal is to twofold:

1. to give as much space as possible for the build team to be creative

2 to realise the vision of the use case specialists in the simplest possible way.


Collaborative Product Development Iteration Workflow
Collaborative Design Session
Adam and Alex meet with clients for Collaborative Design Session
Determine approach
Adam and Alex discuss, determine approach and set parameters for development; requirements vs. areas of flexibility
Initial brief
Alex creates initial brief describing the scope of work in current iteration of development
Coko Design Meeting
Yannis, Christos, Julien, Adam, and Alex meet to discuss brief, development work, and mocks. Pick a target date for completion of current development cycle.
Begin development and mocks
Begin development
Create mocks
Update brief with mocks as necessary
Yannis and Christos begin development on areas that don't need mocks. Julien creates mocks for new interfaces
Final brief and continued development
Circulate final brief. Team continues to discuss approaches to areas that are still flexible.
Timeline: 1 week maximum

The build workflow of the Collaborative Knowledge Foundation (Coko)

During this meeting, the build team can create rough mock-ups on a whiteboard, if necessary, and make UI/UX and code decisions. The key to success is achieving consensus on clearly defined design decisions at every step. When the UI and code people have discussed all parts of the design and have made their decisions, the team works out an order in which to address these issues.

Once the team has made its decisions, the following must occur:

1 update the design brief that was circulated with the decisions made in this session. This brief is the 'single source of truth'.

2 code specialists can begin work.

3 the UI specialists can develop mockups. The code specialists can usually start working on other tasks while waiting for these mockups.

4 the brief must be updated with the finished mockups.

The Build Period

It is pretty simple — the build team then continues until done! This part of the process should not be over-managed and requires little facilitation. Good people, with good challenges, when left to get on with it and with a clear understanding of what is required from them, get the work done. The code and UI/UX specialists should be able to proceed unhindered. They may need to regularly check in with the use case specialists to get further specifications, designs, and the like. Occasionally, the facilitator may need to check in that they have everything they need, and should also look ahead a little and make sure that any of their future needs are anticipated. However, the facilitator should also avoid looking over their shoulders and continually asking 'are we there yet?'

I firmly believe the build team should use whatever tools they choose for managing the process, no need for them to slow down while learning to use the facilitator's favorite tool.

Once the build is complete, schedule another Design Session with the use case specialists to review the current work and to design the next part of the solution.

Facilitating the Build Period

Facilitating the Build Period comes down to the initial meeting between the facilitator and the build team, consisting of the open source project's UX/UI and code specialists. Whether in person or remotely, the facilitator needs to keep in mind that this is when the build team has the opportunity to be creative. During this meeting, the facilitator's primary role is to tease out each member of the build team's ideas, ensuring that everyone is heard and that all their ideas are considered.

This process should be fun and relaxed. Make sure that everyone has a say and that they can be creative with their ideas. In my experience, these sessions need only two hours or so with a small team. It might take longer if larger teams are involved. In those cases, I recommend breaking into small groups and each tackling part of the problems at hand.

During the session, go through the design brief, line by line, with everyone. Add any extra detail that comes to mind which may not have made it from the Design Session to the brief. The facilitator must be careful not to insert their personal opinion on how these things should be approached, and also needs to be very clear on what isn't negotiable and what the group can still work out. Usually, there will be a lot still to be worked out. In the rare case that there *isn't*, this needs to be stated clearly. Nothing is gained except resentment if people are misled into thinking they have more say in the solution than what they actually do.

Open up the conversation at a very high level. An opening question like "So what do you think?" can get the ball rolling. Don't feel a need to jump in and fill in the silence if that's all you get in return. Remember that this session is for the code and UI/UX specialists to have their voices and ideas heard. Most likely, the session will start with some clarifying questions. As a facilitator, you don't need to have all the answers. Be very careful not to throw in your opinion where there isn't (yet) an answer to the question at hand. Be very prepared to say "I don't know" or, better yet, "I don't know, what do *you* think?"

The session should then be as free ranging as possible. Allow people to hammer out ideas. Listen especially to the quiet voices, as I've found that more often than not they are the ones with the winning ideas. Be prepared to explore seemingly off-the-wall ideas and encourage people to bring them forward.

Many logical dead ends may need to be explored, and a number of proposed solutions worked through before identifying the most elegant solution.

Throughout this session, document each point that the group agrees upon. Make sure these agreements are stated in simple clear terms, agreed on by all explicitly, and documented clearly.

There may be times when this session comes up with ideas that contradict what the use case specialists have already decided. This is OK for discussion and exploration, but the agreed solution must not disagree with the design the use case specialists have already decided upon. In these cases, wait for the right moment and simply state that this is not what the use case specialists wanted. Then move the team towards ideas that are in harmony with what was wanted.

There are many technical questions that may arise which affect how the code and UX specialists realise the use case specialists' design.

Don't be afraid to let the UX and code specialists work out a lot of this deeper technical detail during the Build Period. It's good to give them this autonomy and creative space. You only need to concern yourself making sure their decisions won't contradict the design brief. It pays to double check sometimes, too.

If there are clarifying questions for the use case specialists, it is most important to encourage the build team to ask those questions directly to the use case specialists. However, the facilitator may need to keep an eye on some of these interactions to ensure that the build team isn't trying to convince the users that a different solution is better than the one they've designed. The build team must respect and adhere to the design brief at all times.

One very important general rule for these meetings and the consequent decisions is that the build team must realise the simplest possible solution for the use case specialists' design. Don't get into extremely complicated edge cases or open-ended 'what ifs.' Keep the solution as simple as possible.

At the end of this session, immediately update the brief with all of the agreements. The code specialists should be able to start work right away, and the UX/UI specialists may need to work out some interactions and mockups and add them to the brief in the days to come. If this is the case, then don't let that process drag out. Be firm about a two- to three-day timeline after this session to consider the brief 'done.'

Let the build team get on with its work. If the right people are on board then they should give you a rough (and it can be a *very* rough) estimate of how long the Build Period might last. Don't impose a deadline on them. Let the build team give their estimate of timelines. However, expect at all times that their estimates are probably too short! Be easy on timelines, and give the build team as long as they need to do the work.

When the session is over, take the results back to the use case specialists for them to review. Move forward with the next Design Session to solve the next part of the problem.

Onwards

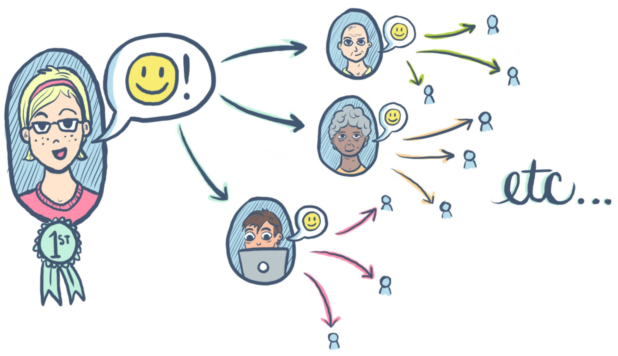
Adoption and Diffusion

The first group of use case specialists, the people that designed the software, will be the early adopters. They're good allies to have because they have buy-in. They'll be enthusiastic and eager but patient when using the software in its early stages. Get them using the software as soon as it is viable so you can all learn and improve the software together.

After several iterations of testing and development, there's a solid application. What's next? It's time to take the solution to the rest of the world.

How can the Cabbage Tree Method be used to migrate a product from the small group of early adopters to a larger base of users? Through diffusion, a strategy for stimulating the adoption of a product into a wider market by taking advantage of the networks of the early adopters.

It's a simple idea, much like adding layers to an onion. The first adopters of the solution can convince others on 'the next layer out' to adopt the product. They're the solution's evangelists. They'll help introduce the product to the next level of adoption. They'll turn their friends and colleagues into users, who in turn become advocates, and so on.



The diffusion strategy has been proven out in the real world. The following example comes from the medical sector, which I learned about first hand from John Abele of Boston Scientific.

Early in his career, John was involved with developing cutting edge (or non-cutting edge, as the case may be) technologies for non-invasive surgery. Today, non-invasive surgical techniques are commonplace. Back then, however, surgery was invasive by definition. Back then, talk of non-invasive instruments for surgery would be like talking about screen-less phones today. Imagine trying to sell that.

Because surgery was defined by 'cutting', the market was hostile to this new idea. So John had a hard time trying to generate adoption for a technology that he knew could transform the medical sector and help millions of people. As he writes:

> “*We were developing new approaches that had huge potential value for customers and society but required that well-trained practitioners change their behavior. … Despite the clear logic behind the products we invented, markets for them didn't exist. We had to create them in the face of considerable resistance from players invested in the old way and threatened with a loss of power, prestige, and money.*”

Smart people who are under the painful burden of outdated technology often resist systemic change. Why? Because it requires them to alter their established ways of working. This, rather normal, resistance to change, can be a huge obstacle to adoption.

In John's case, he drew on some insights he had gathered early on in his career from Jack Whitehead, CEO of Technicon, a small company that had the patent for a new medical device. When trying to bring this product to market, Technicon also had the odds stacked against them. No one, from the lab technicians through to the professional societies and manufacturers, wanted anything to do with it. So Jack drummed up some interest from early adopter types and came up with a surprising next step. He “told all interested buyers that they'd have to spend a week at his factory learning about it.” Further, they would have to pay to attend.

That sounds like an odd sales pitch now, and back then (early 1960s), apparently it sounded a whole lot more crazy. Nevertheless, Jack convinced a handful of excited early adopters to seize that day and brought them into his factory.

During that week, the early adopters were not treated like customers but like partners. They were part of the team. They came to know each other, they worked together, they helped to shape the product further. They *became* the team. Sound familiar? This is pretty much how CTM works. The users become the team.

As John says:

> “*When the week ended, those relationships endured and a vibrant community began to emerge around the innovation. The scientist-customers fixed one another's machines. They developed new applications. They published papers. They came up with new product ideas. They gave talks at scientific meetings. They recruited new customers. In time, they developed standards, training programs, new business models, and even a specialised language to describe their new field.*”

This meeting of once potential customers, now team members, not only contributed to the design of the technology but then took it out into the world and fueled adoption and interest in the product. What had humble roots with a group of early adopters was on its way to creating large-scale change.

John witnessed this process and realised it was essentially strategy, not whimsy: “[Jack] was launching a new field that could be created only by collaboration — and collaboration among people who had previously seen no need to work together.”

John went on to form Boston Scientific and refined this strategy further with Andreas Gruentzig when introducing the balloon catheter to a hostile and uninterested market. Again, he was successful in catalysing large-scale change.

Astonishing.

But, on reflection perhaps there are no surprises here. It is the way open source has always operated. You could have told the same story about any number of successful open source projects. Indeed, as John also reflects:

> “*Just as Torvalds helped spawn the Open Source movement, and Jimmy Wales spearheaded the Wiki phenomenon, Andreas [Gruentzig] created a community of change agents who carried his ideas forward far more efficiently than he could have done on his own.*”

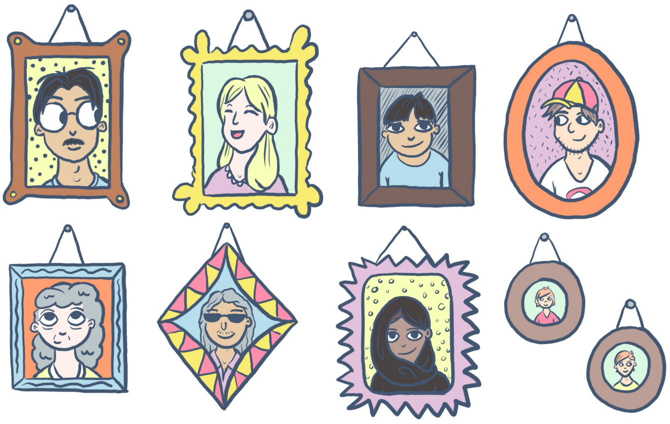
Each of these examples has created change on a massive scale and their success stems from a simple common strategy – to create a community of change agents. John Abele did it with surgical instruments. Linus Torvalds did it with an operating system kernel. Jimmy Wales did it with information. Now we need to leverage these exact same strategies to fuel the adoption of world-beating user-facing open source products.

Diffusion works because the users *are* the community. They feel ownership of the processes and the result. They are the change agents.

It should come as no surprise that a community of change agents is exactly what the Cabbage Tree Method produces in the Design Sessions. You must empower each community member to take the product into the world and convince more people of its usefulness, perhaps even drawing them into future Design Sessions. This is how we fuel adoption, and this is how the product will evolve and improve while continuing to gain a wider base of users.

Things to Think About

Let's Be Friendly



Attribution in technical projects is a fascinating topic. It's fascinating. It's important. And, very occasionally, it's controversial.

The big question is *Who should get credit for a work?* In open source, it's generally accepted that code specialists get attribution for the code they create. Projects give credit by documenting individual contributions to the code base in the copyright and contributions file of an open source project. You can also, of course, look at any code repository and see who has added what over the course of a project.

Attribution for code specialists is pretty clear cut. But what of the other people who are involved in a project? What about the use case specialists, UI/UX specialists, the people who make wireframes, the high-level systems architects, the project or product managers, the ideas people, the founders, and the documentation teams? They don't contribute code, but they all make a contribution to the success of the project. Where and how do we acknowledge their work?

Their stories don't often get told. When those stories are told, they're in narrative form on websites or blogs. Those stories tell a personal or organisational journey about the development of a piece of software from idea to reality. But unlike the contributions files and history records in code repositories, blogs and websites have a much shorter lifespan. Blogs and websites eventually disappear, and when they're lost, so are the stories they've told. When that happens, there's only the record of the code to tell the story. If we believe those remaining records, then we would conclude that the only people who contribute to a successful software project are the code specialists.

That's a pity because all of these varying types of contributions are critical to the lifecycle of any software. Where would any successful desktop software be without the contributions of testers? Many software solutions would not get a second look without a designer's touch or feedback. Some projects would never have been born if it wasn't for the inspiration of some energetic soul who managed to convince others of the value of an idea.

A Personal Example

For almost two decades, I've been fortunate enough to work on many successful and interesting technical projects. I've often been the ideas guy. I'm neither a developer nor a designer. I suck at QA. I do, however, major in musing on the possibilities that technology can offer.

Sometimes, I'm fortunate enough to receive credit for this work. I cherish those few moments when people have recognised my contribution and have credited me in the occasional blog post. Martin P. Eve (a good friend and all-round solid guy), for example, recently wrote this in a post about his JavaScript Typesetting Engine CaSSius:

> “*I can't remember when Adam Hyde first suggested to me that CSS regions might be a viable way to produce PDFs for scholarly communications but it seemed like a good idea at that time and, I think, it still does. CaSSius is my implementation of that idea.*”

Without this mention, there would be no other record of the effect I had on Martin's thinking and practice. I was humbled by his generosity.

When these things happen, even though they rarely do, it motivates people like me. It puts some fuel in our engines to keep moving on and to continue working the way we do.

Sometimes, however, it goes the other way. Occasionally, and once again (thankfully this time) it's rare, there are those who believe non-technical contributors shouldn't receive any credit. Once or twice someone has been outraged when I casually mentioned my involvement in a project that I managed, initiated or inspired.

Many of the contributions I've made have been made to small projects. Small projects that represent not much more than a line item in a long career of coming up with innovative approaches to many interesting problems. Over time, I've come to understand that for some people, the line item is the source of great pride. This may be the only innovative project that person has ever worked on. While I’ve begun to understand it, this kind of possessiveness doesn’t make a whole lot of sense in the open source world.

Attribute Generously

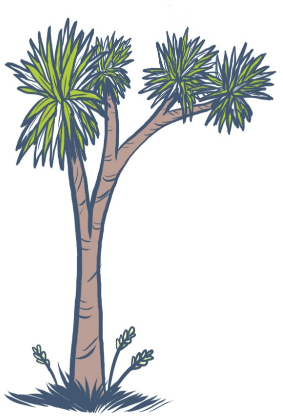
The good news is that many open source projects **do** offer attribution for many types of work. Mozilla, for example, has a single global credits list that records the names of people that have made “a significant investment of time, with useful results, into Mozilla project-governed activities.” Audacity is a favorite project of mine that credits pretty widely, categorising contributions to include code, documentation, translation, QA, administration, and the like. These are great examples of software projects which recognise and honour a variety of work.

We should all follow the examples of Mozilla, Audacity, and other projects like them. Open source can only benefit from the attitude towards attribution that these projects embrace.

We need to think carefully about how technical projects value and attribute work. Architects often proudly say “I built that house.” Did they actually lift the hammer and cut the wood? Probably not, but I think they have a right to proudly state their contribution as much as users, designers, developers, managers, QA folk, and ideas people have a right to state and be recognised for the contributions they made towards a software project.

There are far too many folks whose contributions go unacknowledged. We should all celebrate and recognise the large variety of contributions that go into creating shared open source solutions and find effective, lasting, ways to tell those stories.

Cabbage Tree Open Source



The Cabbage Tree Method, as you've seen, is a design-build methodology. But its intended impact isn't merely to improve the process through which we can produce amazing user-facing open source software. The Cabbage Tree Method advocates a different kind of culture in open source projects, a culture that's inclusive and which requires, and in fact celebrates, a wide range of skills and roles, a culture that also places the use case specialist at the centre of the process to design a solution.

This kind of shift in culture may not be an easy thing to achieve for either a single project or the open source movement as a whole. There are those working in open source who aren't at all tolerant of new approaches. After posting my ideas to an open source foundation's mailing list, for example, I was taken seriously by some, but ridiculed (and worse) by others. More than once I was advised that only developers should start open source projects. Any other alternative was labelled 'unwise'. I was also told that free and open source projects shouldn't need to work with designers. I was informed that my statements pointing out the dominant demographic being 'white men that know how to code' was a result of my unconscious racism. Not many addressed the important points I hoped we would discuss, points like the fundamental misunderstanding of the value of the 'itch to scratch' model that open source holds so dearly.

I'm not a coder. However, I have started many successful open source projects and I will start many more. I don't believe open source is about code (even though some of the people I respect most in the world are "white men that know how to code"). I believe open source is about solving problems with, and as, a community. That means tackling problems developers have, and tackling problems that everyone else has. I believe the fundamental strength of open source is making the 'people with the problem' the driving force in the solutions team. I believe that this means open source culture needs to examine its current modus operandi and come to terms with its failures, address some of the issues, and experiment with new models for projects and cultures within projects.

I believe there is a great need to diversify open source models.

From experience, I also know that starting an open source project is the most important culture-setting moment you will ever have. Starting projects in new ways leads to new models. I firmly believe these new models are the way forward for open source into areas where it's not having much success.

I want to suggest to anyone out there who cannot or will not write code that **you** are the future of open source. Your vision, by virtue of the fact you do not write code, is exactly what open source needs to diversify cultures and methods. You need to bring your ideas to the table, whether that's to an existing project or as the ignition point for a new one.

Unfortunately, there are few examples that show 'non-coders' [sic] how to start a new project. You need to work it out for yourself. While you're doing that, trust your instincts. Find a way to make it happen which is consistent with your ideas and your vision. That is what being a pioneer is all about. My advice on this, based on involvement in many open source projects, is that I've found success in operating in good faith and by entrusting others with my vision. It's crucial to infect others with the excitement of the mission. That process, as it happens, is also a fundamental tenet of open source: start with a common itch, and build community to scratch it.

I'm proof that it can work. Don't listen to anyone who tells you that starting an open source project is a bad idea. Go ahead and make it happen. Give yourself permission to be a little bit stubborn and to go against the grain. Then, make sure you let others know about your experiences so we can all learn from your successes and failures just as, I hope, you might learn from some of mine.

Colophon

Adam Hyde — *https://www.adamhyde.net* , *https://coko.foundation*

Scott Nesbitt — *https://scottnesbitt.net/contact/*

Pepper Curry — *https://peppercurry.com*

Raewyn Whyte — *http://allmyownwords.co.nz*

Julien Taquet — *julien@lesvoisinsdustudio.ch*

Cabbage Tree Method — https://www.cabbagetree.org

The text font of the book is Vollkorn, designed by Friedrich Althausen, *http://vollkorn-typeface.com*, while headers and caption are set in Cooper Hewitt, designed by Chester Jenkins, *https://www.cooperhewitt.org/open-source-at-cooper-hewitt/cooper-hewitt-the-typeface-by-chester-jenkins/* .

This book was produced using the open source software — Ghost, WordPress, Booktype, Editoria, and Vivliostyle. Interestingly this book on the Cabbage Tree Method was created with Editoria, and Editoria was created using the Cabbage Tree Method!

The first edition of this book was printed by Edwards Brothers Malloy at the end of January 2017, in San Francisco, United States of America.

Collaborative Knowledge Foundation

SHUTTLEWORTH FOUNDATION