Samman
Coaching

Technical Agile Coaching with the Samman method

By Emily Bache
Foreword by Kent Beck

Technical Agile Coaching with the Samman method

Emily Bache

This book is for sale at http://leanpub.com/techagilecoach

This version was published on 2023-02-16

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

Tweet This Book!

Please help Emily Bache by spreading the word about this book on Twitter!

The suggested hashtag for this book is #coachingsamman.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

#coachingsamman

Also By Emily Bache

The Coding Dojo Handbook

Mocks, Fakes and Stubs

Contents

Introduction

There are 5000 lines of code in the file of code that appears on the screen in front of you. One of the team members, Lars, explains he has had several bug reports connected to a few of the private functions defined here. Lars would like there to be some unit tests for this code. Time to roll up your sleeves and get stuck in! You spend a few minutes discussing possible approaches with the team and decide to try a technique they haven't used before. The ensemble gets rolling with you initially in the navigation role. A couple of sessions later everyone is taking their turn and the code is starting to be covered by tests. Lars comments he likes the feeling of freedom to change the design and get quick feedback in the form of passing or failing tests. It should be easier to find and fix bugs in this code now.

At the same time you are also working with a second team. Their task is to write a completely new piece of functionality. Patrik, (a team member), has done some experiments, sketched out an overall approach, but he hasn't written much code or any unit tests yet. People in this team don't normally use tests to help drive their design and they are curious to try it. You facilitate a whiteboard discussion to create a rough test list and articulate an initial goal. You set up the ensemble timer and ask Patrik to become the navigator and begin writing the first test. With some prompting from you, he makes a good start. When the timer pings he hands over to another team member. A couple of rotations later everyone can see the test both compile and fail. Success! The first step of TDD is complete. Making the test pass turns out to be straightforward, so the next navigator soon moves on to write the next test. Over the course of a few sessions Patrik and everyone in his team will experience doing Test-Driven Development in their production code. Patrik comments it was actually fun to write the tests this way instead of afterwards.

One hour each day is set aside to do some practice and learn new techniques. In one of these "Learning Hour" sessions both teams are gathered and you have prepared some materials about the "Golden

Rule of TDD". You ask the group to call out five important things to remember when doing Test-Driven Development. This is a quick warmup to remind everyone what they already know about TDD, and you write up what they say on the whiteboard. Nobody quite expresses the golden rule although one person was close. You explain the rule: *Don't create any production code without a failing test that requires it!* Next is the most fun part - writing code. Working in pairs on a simple exercise, everyone tries to apply the golden rule. When there are a few minutes left of the session, we stop coding to discuss what happened. Several people found themselves writing the code first by mistake. One person was doing well then wanted to create new code in a refactoring step. Was that ok? A short discussion ensues. At the end of the session everyone goes away with a sticky note or a screenshot of the rule and homework to try the exercise again by themselves.

Does this way of working sound interesting to you? It's called the Samman method and it's a way of doing Technical Agile Coaching. The rest of the book explains all about it.

Samman Technical Coaching

Samman Technical Coaching is a method for people who want to make a difference and improve the way software is built. The focus is specifically on technical practices and how people write code. The foundation of it is cultivating good relationships with the people you work with. The rest is about effective ways to learn from one another and to change behaviours for the long term.

A Samman coach divides their time between several software development teams and there are two main parts to the method:

- Learning Hour
- Ensemble Working

In the learning hour the coach uses exercises and active learning techniques to teach the theory and practice of skills like Test-Driven Development and Refactoring. In the Ensemble sessions the whole

team collaborates together with the coach in applying agile development techniques in their usual production codebase.

I chose the word 'Samman' to describe this coaching method since so much of it involves working in an ensemble. "Ensemble" is a French word meaning "together". "Samman" is a Swedish word that also means "together". I wanted to distinguish this method from other ways to do coaching and make it easier to search for on the internet.

Why would an organization engage in Samman Technical Coaching?

What's in it for them? For the organization investing in coaching? Well ultimately, increased business agility and success. If a company is building a software product, then the way people write code actually matters quite a lot. Specifically:

- Good technical practices mean the organization can build new features with a shorter lead time and higher quality. That should mean meeting deadlines and delivering reliable software. This brings happier customers and a successful business.
- Skilled developers will want to work for an organization with high quality code, effective development practices and a healthy culture.
- The company needs to avoid drowning in technical debt. If it piles up too much developers will ultimately be unable to deliver any new features in a timely or cost effective manner. That is a serious risk for any business.

There is well respected research that shows exactly these effects. The book *Accelerate* by Forsgren et al. explains the remarkable findings from a multi-year study of software development organizations. They studied many factors, and identified those that contribute most strongly to business success. In particular they highlight the practice of Continuous Delivery. That umbrella term includes several technical and organizational practices, including Test-Driven Development.

In follow-up research[1] the same authors also found that technical debt directly reduces developer productivity.

Why would you choose to coach using the Samman method?

What's in it for you as coach? Well, it's the best method I know for improving the code that gets written. Life as a developer is more fun when the quality of the design and the tests is high. It's easier and faster to add functionality, please the users, meet your commitments, keep your boss happy, all of that. Plus when the code is well-written and the design is flexible you're more likely to get to introduce fun new tools and technology.

Those are all benefits of the end result - higher quality code. Benefits for you personally doing the job as coach:

- It's challenging and interesting. You never know what code the team is going to put up on your screen, and you never know how people will react when you ask them to work as an ensemble, do TDD, refactoring etc. There is a lot of variety in the interactions and the kinds of problems you face.
- You can have a bigger effect than you would in other roles. You get to influence several teams over a few weeks of coaching. Over months and years you influence several organizations.
- Teaching is inherently rewarding - witnessing someone start to understand a new concept and recognize how they can apply it in their work. That feeling when you see that happen. Gold.

I can't cite statistically sound research to prove these points, unfortunately. I just encourage you to try it for yourself and see if it works. You should see improved code and happier teams following your coaching. The other important question is whether you enjoy working this way. I know I do.

[1] https://cloud.google.com/devops/state-of-devops/

Elevator pitch for Samman Technical Coaching

In order to be successful in modern development organizations, software developers need new skills. These skills are not easily learnt on a short training course or at a university. Practices like Continuous Integration and Test-Driven Development require developers to change their minute-by-minute habits and ways of working.

As with any new skills, the way to learn is through a combination of teaching and practice. The Samman method includes a series of short lessons called 'learning hour'. The coach uses active learning techniques that are proven to be more effective than lectures. We work together on code katas and other exercises so developers both *understand* the new techniques in theory and *experience* them in carefully chosen examples.

The Samman coaching method is also on-the-job, together with developers in their usual situation. In order to change the way developers work it's often not enough to only practice on code katas and learn theory. The coach works together with development teams in a structured collaboration called an 'ensemble'. We learn how to apply relevant techniques in the actual production code the team works with daily.

Day in the life of a Samman Technical Coach

Right here at the start of the book I want to give you an idea of how a typical coaching day could be structured. This is a sample agenda for coaching 2 teams, in Sweden, where office hours are usually 8-17 and lunchtime is generally 11:30 or 12:00.

08:00 Arrive, check messages, prepare for today's sessions
09:30 Ensemble working with first team
11:30 Lunch
13:00 Learning Hour with both teams
14:00 Short break
14:15 Ensemble working with second team

16:15 Debrief, reflect, write a summary report
17:00 Go home on time!

As you can see, the bulk of your time is spent interacting with people and writing code: **Ensemble Working**. The next main item on the agenda is a one hour teaching slot, the **Learning Hour**. The rest of your day is spent preparing for those sessions as well as communicating with others in the organization so the whole coaching engagement runs smoothly.

The great privilege of a Samman coach is getting to write code for most of the day, together with others. You work in an environment where everybody is contributing and learning new skills. You're using effective technical practices and writing excellent quality code and tests together.

What is in this book

This is designed to be a useful handbook for someone working as a Technical Coach using the Samman method. This introduction gives you an overview. The next chapter goes into more detail about the purpose of the coaching - what development practices a Samman coach promotes. The rest of the book is structured in three parts:

- **Part 1**: Coaching in an Ensemble - why and how.
- **Part 2**: Learning Hours - how to plan and execute them.
- **Part 3**: The overall coaching engagement and your career as a coach.

I have deliberately gone straight into some detail about ensemble working in the first section of the book. I wanted to start with the most exciting, challenging, *interesting* parts of being a Samman technical coach. I'd like you to get an impression of what it is like to do this job. Perhaps a more logical place to start would have been 'what kind of person becomes a coach', 'what skills do you need' and 'how to get started'. That material is in Part 3. It's important and useful information, but first let me tell you stories about what it's like when you've got there.

Of course I won't be offended if you choose to read the parts in another order than I've set them out here. If you're not thinking of becoming a coach yourself but perhaps want to hire one, Part 3 might be more interesting for you than Part 1. Choose the starting point that works for you.

How this book relates to my other books

I've been a professional software developer for over 20 years, and during that time I've learnt from a lot of great people. I have also spent a fair amount of time teaching others, particularly in a forum called 'The Coding Dojo'. In 2011 I published "The Coding Dojo Handbook"[2] which is full of advice and experiences to help other programmers to create such a forum. That book has been moderately successful, and many programmers have used it to set up coding dojos and improve their skills. Now, a few years on, I'm feeling the need to write another book.

You can see this work as an expansion of some of the ideas and methods I wrote about in 'The Coding Dojo Handbook'. I've learnt a lot in the roughly 10 years since I wrote it. You don't need to read that book to understand this one, I aim for them to be complementary to one another.

In 2011, I was full of ambition to write a book called "Mocks, Fakes and Stubs". Unfortunately I never found the enthusiasm needed for finishing it, and I don't know if it will ever be finished. What I have found enthusiasm for though, is continuing to write code, practice on Code Katas, and teach others. I've been pleased to work with groups of developers in many contexts, from evening user-group meetings to company trainings, even an accredited course for an adult education college. I believe all programmers need to continue to develop professionally, and I have been finding more effective ways to help with that. This book is an attempt to write some of them down.

[2] https://leanpub.com/codingdojohandbook

Acknowledgments

The main ideas in this coaching method were originally developed by Llewellyn Falco, and I continue to learn a huge amount from him. This all started in 2018 when I did some pair-coaching with Llewellyn and I was impressed with the results he was seeing with his teams. (You can read about that story towards the end of the book in the section "How I became a Technical Coach"). I have adapted Llewellyn's ideas for my situation and further developed several aspects of the coaching method. I came up with the name "Samman Coaching".

I'd like to thank Llewellyn Falco for so freely sharing his knowledge and methods with me. I'd like to thank all my colleagues at ProAgile for all their support and inspiration. I'd also like to thank all the early readers of this book who gave me feedback, including Samuel Ytterbrink, Olof Bjarnason, Alexandre Cuva, Per Hammer, Grzegorz Gałęzowski, Dave Nicolette, J.B. Rainsberger, Ester Daniel Ytterbrink, Bill Wake, Jo Van Eyck, Joe Wright, Ted M. Young, Dianing Yudono, Lars Eckart, Clare Macrae, Heidi Helfand, Philippe Bourgau, Sam Cranford, Thomas Sundberg. I really value you taking the time to help me write a better book.

The purpose of Samman Coaching

A Samman coach aims for *improved development practices* in the teams they work with. This chapter explains what those practices are and what outcomes to expect from the coaching.

Development techniques

The choice of these particular development techniques shouldn't be controversial, they have been used for years by Agile practitioners and in DevOps communities. There are many books and training videos out there explaining how to do them. Samman coaching is a more effective method for introducing them. The next sections go through the main practices we aim to introduce and promote.

Better unit tests

Developers should deliver a suite of automated unit tests together with the code they write. The tests demonstrate each part of the code works as the authors intended, and documents their assumptions. The tests provide a safety-net for refactoring and make maintenance less costly.

In many organizations there are explicit processes in place to ensure that tests are written. However, many developers still struggle to write *good* unit tests. Not many have received any training or feedback about what tests should look like. In the worst case the tests can actually increase maintenance costs without providing a useful safety net.

Continuous Integration

With the rise of Agile methods, Continuous Delivery and DevOps, development schedules are being compressed and teams are expected to deliver working increments of software more frequently. In my experience, the way to confidently deliver a new version of the software every 1-2 week iteration is to integrate and test the software at least every day, so you always have something ready. If you want deliveries even more frequently then you need a corresponding decrease in the length of time between integrations. In general it means the developers should work only for a few hours between synchronizing and integrating their code together.

In many organizations the developers are used to working alone in a branch for days or weeks before integrating their work. They don't know the incremental design and refactoring techniques you need to be able to integrate more often than that. In the worst case everything looks fine until shortly before the intended release date, when big-bang integration happens with a string of bug-causing, schedule-delaying merge conflicts.

Safe refactoring in legacy code

Developers often find themselves working with code that they don't understand very well but that they need to change. In this situation it's difficult to maintain the overall health of the design and developers are tempted to make small kludgy fixes. In the long run this just makes the problem worse. The code becomes more and more difficult to work with and developer productivity sinks. In the worst case the organization has to throw this code away and start again.

There are techniques which can prevent this decline. There are safe refactorings and migrations that developers can use to get difficult code under control. There are ways to add regression tests without taking big risks. If developers know the techniques and can use them skillfully then you can avoid the need for a costly re-write.

Incremental and Iterative Development

These days you don't plan the whole software up front. You deliver an increment, get feedback and adjust your plans. In the code, developers need to use iterative and incremental techniques to develop the design. You can't plan the whole architecture up front.

Many developers struggle to iterate the design of the code. They don't know the refactoring tools, the design patterns or the signs to look for that a different design is needed. These are teachable skills. It is possible to maintain a long-term cadence of regular incremental releases each with high quality. You need great automated tests and a culture of constant design improvement.

Levelling up a whole team together

When I first learnt to do Test-Driven Development and Continuous Integration I was working in a small team on a greenfield development project. Our culture and ways of working were aligned and we were very effective together. Unfortunately that product was not successful in the marketplace and I subsequently moved to another team in the same organization.

The new team culture was very different. I continued to write unit tests but my teammates did not. I continued to work in small increments with frequent integration but my teammates did not. I quickly realized this was not sustainable. The tests I wrote were ignored or even deleted by my colleagues. I was reprimanded for making refactorings that caused merge conflicts for others later on. I was unhappy. They were unhappy. It didn't last - I found myself a new job.

Software development these days is a team sport and it doesn't work to only train individuals. Samman coaching aims to create a whole-team culture shift. In the ensemble working we discuss the minutiae of how to use new techniques in the particular situation the team finds themselves in. We build consensus about how *this team* wants to work. In the learning hours we discover what development *could* be like if we used new ways of working. The team becomes aware

that a different future is possible. The coach helps them gain the skills needed to go there.

Expected outcomes

Firstly, we aim for awareness of what good unit tests look like, what continuous integration actually is, that it's possible to safely refactor code you don't initially understand. Next we aim for the insight that successfully meeting deadlines and delivering high quality code means learning iterative and incremental development techniques. Teams gain new aspirations to be good at these things.

In my experience after perhaps 10-20 coaching days most teams will have reached those insights. These are the first steps on the road to improving the way that team works and the quality of the code they deliver. After that, it's about building skills and anchoring ways of working in habits and culture.

I first learnt about the methods explained in this book through pair-coaching with Llewellyn Falco. I was impressed with the results he was seeing. The outcomes I observed were firstly a clear attitude change. People wanted to learn these techniques, they saw them as valuable. Secondly there was one team in particular where they had become really skilled. Over the course of many days and weeks with the coach they had changed the way they designed and built their software. I remember sitting at the back of the room observing this team working as an ensemble. They were working together smoothly and productively, creating code and tests in small increments with high quality.

Those are the outcomes you should see from Samman coaching. Changes in attitude, increases in skill, improved code quality. Over time, more productive development teams.

Measuring outcomes

Most people like to see hard evidence that something works before they spend a lot of money on it. Or at least, they might let you start

coaching based on your reputation but soon afterwards they will want evidence. Numbers help with that.

The first thing to measure is attitudes. A simple survey should suffice to show that after the coaching developers are more enthusiastic about using these techniques. After that you expect to see improved code quality, increasing number of test cases, more frequent integration. Hopefully you can track those kinds of things by examining your existing development infrastructure.

After a while of coaching you hope to see teams meeting deadlines more reliably, reductions in production bugs, and fewer calls from despairing developers who want to re-write the whole system from scratch.

Ultimately you'd like to observe increased productivity. Unfortunately it's really difficult to measure the productivity of software developers. A great deal of research has been done. I recently read "Rethinking Productivity in Software Engineering", a compendium of essays from a number of researchers summarizing the state of the art in 2019. My conclusion from reading it was that it *might* be possible to measure productivity of software developers, but that it is *really difficult* and probably outside the reach of most organizations I work with.

You also need to be aware of Goodhart's law. Anything you measure and start to use as a target can become *gamed*, (ie people work to get better numbers instead of improving anything important). Some measurements are more susceptible to that than others. Any numbers you gather need to be backed up by qualitative measures - talk to people and find out if they think things really are improving or if it just appears that way.

Part 1: Ensemble Working

As a Samman technical coach the majority of your day is spent sitting with teams working as an ensemble. That is:

"All the brilliant minds working together on the same thing, at the same time, in the same space, and at the same computer - We call it 'Mob Programming'"

– Woody Zuill

That quote is from "Mob Programming - a Whole Team Approach"[3] by Woody Zuill and Kevin Meadows. I originally learnt the technique from Woody and others close to him. Over the years since then I've come to prefer other words to describe it. I usually say "ensemble" instead of "mob". The essence of the technique is the same though.

I like the way the word "ensemble" implies friendly people collaborating, like a group of musicians. It seems to me to much better describe what this activity is actually like in practice than the word "mob". In Sweden, (where I live), "mobba" actually means "to bully". It's not a good association to make. Another change is I also prefer to say "typist" instead of "driver". I like to make it clear the person with the keyboard is not in charge of the direction the group takes.

A note on pronunciation. "Ensemble" is originally a French word that has been loaned into many other languages. In English I pronounce it "on-som-bull". Your own language may have another way to say it.

[3] https://leanpub.com/mobprogramming

This section of the book goes into some detail about what a Samman technical coach actually does in the ensemble sessions, and why it's such an important part of your work.

Ensemble Primer

For those of you who have not experienced working in an ensemble, a short introduction could be helpful here. Many programmers have tried pair programming, and conceptually, working in an ensemble is the same but with more people. You still have only one computer being used to write code. Everyone present is responsible for the code being produced, even though only one person at a time is typing. However, for an ensemble to work smoothly you need more structure and roles than you generally have in a pair. There are three main roles, and you rotate roles frequently.

Typist

The typist is the person who has the keyboard and mouse. You use the development tools and operate the computer. The important rule here is that you are not allowed to decide what code to write or what tests to perform. The typist listens carefully to everyone present, and most particularly to the navigator. You enter the code the ensemble has decided on, to the best of your ability. The navigator is usually the spokesperson for the ensemble, but the typist may also follow advice from other people. You can always ask questions to clarify what is wanted and ask for more detail.

The typist should be in full control of the editor, the testing tools, the debugger, the commandline, the refactoring shortcuts - everything to do with operating the computer. Things like entering new code, browsing through existing code, running tests and making commits.

Navigator

The navigator speaks for the ensemble and explains to the typist what code they should enter into the computer. They speak in words,

out loud, explaining the development activities they have in mind in enough detail that the typist knows what to do. In this way, everyone in the group both hears the navigator explain the work, and sees the typist do the work.

At first this feels very strange - we are not used to talking about code in this way. Many people don't actually know the higher-level vocabulary for talking about code. An inexperienced lead might say things like "public int foo parenthesis int bar close parenthesis" when they'd probably be better off saying "define a function called foo. It takes one argument, an integer called bar". You are talking to the typist, not to a computer! They generally respond well to high-level descriptions.

Team-members

Everyone who isn't the typist is a co-navigator in the ensemble. We all develop the software together. The aim is that the whole ensemble should instruct the typist with one coherent voice. Usually that means appointing someone to "be the navigator". They lead the work, they are talking and making the detailed decisions. Everyone else is in a supporting role, quietly waiting for an opportunity to contribute. With a more experienced ensemble, this leadership role may not be explicitly appointed, it could simply pass fluidly between members without any formal handovers. People chip in when they have something valuable to contribute, and stay quiet otherwise.

Teammembers support both the typist and the navigator. If you know a good keyboard shortcut the typist doesn't seem to be using, you can suggest it. If you can see a way to reduce duplication or improve design, you can suggest it. The important thing is to choose carefully when and how to make the suggestion. You don't want to distract or cause context-switching or confusion. If the typist and navigator are getting along well doing something else, wait with your comment.

Choosing your moment to speak and knowing when it's better to stay silent is a really important skill. Having said that, it is important that every member of the ensemble should follow what's going on. You should always feel free to ask questions so you understand the code that's being written. Just pick your moment wisely.

Group Discussions

There will be periods when no code is being entered, and instead we are discussing what to do. This is a normal part of coding work: we need to agree on a goal, an approach, a design, a next test case… Many discussions become more productive when you all stand around a whiteboard or shared online document and can collaboratively sketch your ideas. During these discussions you pause the ensemble roles and let everyone take part equally. It's wise to have a bias to action - don't discuss too long before returning to the ensemble and writing some code together.

Facilitator

It's worth remembering that working as an ensemble is a skill that takes some time for a group to learn. It helps to have a facilitator whose job it is to keep things working smoothly. This is usually someone different from the typist or navigator since it's hard to facilitate at the same time as doing one of those other roles. This person will spend their time reminding people of the working agreements, making sure roles rotate regularly, and helping the ensemble to reflect and improve their work together.

As the ensemble gains experience the facilitator may need to intervene less often. It may become a part-time position or disappear altogether. To be honest, I haven't worked with many really experienced long-term ensembles and I'm speculating here.

Rotating roles

Most ensembles have some kind of timer that alerts the group when it's time to rotate roles. Many build their own - it's a fun little piece of software to implement. You can customize it with your own alerting mechanism and decide exactly how intrusive it should be and what it should suggest. Many such tools have a list of names and prompt the next people that it's their turn to be the navigator and typist. Other groups have a more informal switching mechanism.

Experience from many ensembles suggests it's a good idea to ask the person who is currently navigating to become the typist when you rotate. As typist they have to stop leading the ensemble, and someone else naturally then finds it easier to step into that role. Other ensembles prefer to do it the other way around - the typist becomes the navigator. Typing gives them a good introduction to what's going on and they can seamlessly take over when it's their turn to navigate.

Other ensemble roles

As well as typist and navigator, there are other roles that people can take as the need arises. A common one is "Researcher". If the ensemble gets stuck because no-one remembers the exact syntax or library function to use, someone can offer to research it. While the ensemble continues with some other task, the researcher gets onto another computer and searches the internet. When they have found something useful they could paste a link into a shared group chat, or otherwise share it with everyone. The researcher can then take up their normal role as an ensemble member and explain what they found out when the group asks for it.

Another useful role to have is "Archivist". It can be helpful to have someone writing stuff down - decisions the group makes, alternatives we looked at, designs we're following. In particular the archivist might keep track of the current goal the ensemble is working towards. It could be written on a whiteboard in the working area, a shared online document or noted in a ToDo list in the ensemble's timer.

The Coach role

Now you have a basic understanding of what working in an ensemble entails, the rest of this section of the book explains how you as a coach can use this forum to promote better ways of working in the code.

Final Thoughts

Before I close, a few last thoughts about Samman technical coaching.

What is the most important thing to remember?

The most important thing is that they like you. This is probably true for all consulting. They won't listen to you if they think you're a moron, or a nasty person, or a fake. Cultivating good relationships can be hard, but it's crucial.

Where can I find out more?

You'll become a better coach if you spend time reflecting on your work and explaining what you're doing to other people. Also, listen to other coaches explain what they do and find out what works for them. You will hopefully find inspiration and incorporate new ideas into your work. You could seek conversations with many different people, for example at conferences, online ensemble programming sessions or internet discussions.

As well as more general conversations, I've found it very valuable to find one or two trusted peers who I talk with regularly. Other technical coaches doing the same work as me but in different organizations. We can talk confidentially and compare notes. This is something I've personally found very important and helpful for my career.

I would like there to be a global community of Samman technical coaches, exchanging ideas and experiences. I've started a website sammancoaching.org[4] which I hope will become a resource for this community. At the time of writing, I don't yet know what this will

[4] https://sammancoaching.org

turn into. If you've read this book then perhaps you're interested enough to check out the site and find out what's happened since the book came out.