THE CODING DOJO HANDBOOK

*a practical guide to creating a space where **good** programmers can become **great** programmers*

Emily Bache

Foreword by Robert C. Martin

The Coding Dojo Handbook

a practical guide to creating a space where good programmers can become great programmers

Emily Bache

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

This version was published on 2013-10-15

ISBN 978-91-981180-0-1

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.

Contents

Foreword

Do you remember this old joke? A young man is on the subway, carrying a guitar case. He's a member of a band that is performing a concert at Carnegie Hall; and he's running late. He dashes off the train and up the stairs, and realizes he's lost. He knows that the performance hall is close, but he doesn't know the direction. So he stops an old man on the street and asks: "Excuse me sir, but how do I get to Carnegie Hall?" The old man looks at the lad with his guitar case and says: "Practice son, Practice."

It is a fundamental truth that all professionals practice. Of course professional musicians practice; and so do professional athletes. Lawyers practice – they rehearse testimony and closing statements. Doctors practice, on cadavers, dummies, and even suturing oranges. All professionals practice.

What do we, programmers, do to practice? We write code of course. Lots of code. We write code at work, and then many of us go home and write more code. We do this because we love writing code; it is a passion for us.

But not all forms of practice are equal. Some ways to practice are better than others. Professional athletes practice games, but they also practice drills. Musicians practice their performances, but they also practice scales and etudes. These other forms of practice are designed to emphasize, and therefore improve, certain skills – especially those skills that are hard to acquire and easy to lose.

That's what this book is all about – a special way to practice that emphasizes certain skills that are hard for programmers to acquire and easy for them to lose. Those skills include working together as a team, the disciplines of Test Driven Development and Refactoring, good design skills, and many others.

In this book Emily Bache describes one of the most popular activities to come out of the Software Craftsmanship movement; an activity that is sweeping across our industry: The Coding Dojo. Based on a martial arts theme, the Coding Dojo is a meeting in which enthusiastic software developers, intent on self improvement, engage in purposeful practice for the purpose of refining their skills.

The Dojo gives a formal structure to such practice. That structure is complete with rituals, disciplines, rules, and procedures that promote effective learning and minimize distraction. The Dojo is a safe place to practice with, and learn from, others. Best of all, the Dojo is *fun*!

In this book you'll get a feel for just how much fun this can be; because Emily avidly describes the fun *she* has had in setting up, running, and participating in Dojos. Her enthusiasm is contagious. You'll read about her adventures, successes, failures, and just the overall great time she's had while learning, and helping others to learn, in the Dojo setting.

With her lively and readable style, Emily teaches us how to set up a Dojo, and what the principles, rules, and procedures are. She tell us how to deal with what she calls: "Dojo Disasters"; and she describes the various forms of practice such as Kata and Randori. And, perhaps most importantly, she provides a catalog of the exercises that she has found most beneficial in a Dojo.

But there's more to this book than a description of Dojos. While describing the disciplines and principles of Dojos, Emily also engages us with a very cogent and enlightening description of some of the most important software disciplines of the last decade. These include Test Driven Development, Refactoring, Continuous Integration, Test Automation, and many, many others.

In short, while this book is a wonderful workbook for practice; it is also a tutorial in *what* to practice. The subtitle says it all: This book is about making good programmers great.

Is that your goal? Do you want to be a great programmer? Then you don't want to miss this book. Because to become great, there's only one absolute rule: Practice child… practice.

Robert C. Martin

17th November 2012

Introduction

As a professional programmer, how do you learn new skills like Test Driven Development? Pair Programming? Design principles? Do you work on a team where not everyone is enthusiastic about good design and writing automated tests? How can you promote good practices amongst your colleagues?

I've worked as a programmer for many years, and these kinds of questions have come up again and again. This handbook is a collection of concrete ideas for how you can get started with a coding dojo where you (and your team) can focus on improving your practical coding skills. In my experience, it's a fun and rewarding activity for any bunch of coders.

Learning new skills inevitably takes time and involves making mistakes. In your daily work environment where the focus is on delivering working production code, it can be hard to justify experimenting with new techniques or to persuade others to try them. When I attended my first "Coding Dojo" with Laurent Bossavit and Emmanuel Gaillot in 2005, I could see these kinds of meetings could be a fun way to effect change.

When you step into the coding dojo, you leave your daily coding environment, with all the associated complexities and problems, and enter a safe environment where you can try stuff out, make mistakes and learn with others. It's a breathing space where the focus is not on delivering solutions, but rather on being aware of what you actually do when you produce

code, and how to improve that process. The benefits multiply if you can arrange to bring your whole team with you into the dojo. Through discussion and practicing on exercises, you can make a lasting impact on the way you work together.

Following the dojo I attended in 2005, I brought Laurent to my (then) workplace to show us all how it was done, and from there I began to facilitate coding dojos in various other settings. I've done them with my immediate colleagues, user groups, at conferences, and more recently as a paid consultant brought in to do training with teams. Inspired by Corey Haines, I've also led "Code Retreat" days, which is a kind of scaled up coding dojo. All these events have been good fun - coders enjoy coding! We've had excellent discussions, learnt from each other, and written a significant amount of clean code and tests. It seems to me that acquiring skills like TDD, Refactoring and pair programming is a long process - it takes years - and it is a lot more fun and rewarding if you can get a like minded group of people to join you on that journey.

This handbook is a collection of practical advice drawn from my experience, with concrete ideas for how you can get started with your own coding dojo. There is a catalogue of "Kata" coding exercises that you can try, and advice about how to choose one for your particular situation. There are many useful resources on the internet which you can use to augment your dojo, and some are reviewed here.

Kent Beck once said *"I'm not a great programmer, I'm just a good programmer with great habits"*[1]. What are you doing to improve *your* coding habits? This is the book with the advice and encouragement you need: get together with some like-minded people and hold a coding dojo! It's fun!

[1] page 97 of "Refactoring" by Martin Fowler

Acknowledgments

This book has its origins in the work of Dave Thomas, who introduced the idea of the Code Kata, and Laurent Bossavit who came up with the idea of the Coding Dojo, and co-founded the first one in Paris. Over the years many others have also contributed to develop the idea and the practice. I'm especially grateful to Laurent Bossavit, Emmanuel Gaillot and Fredrik Wendt, pioneers who I have collaborated with and learnt from in the dojo.

Over the years I have met many people in coding dojos, and I am grateful to have learnt so much from them. There are some I have met in the dojo who I count myself particularly lucky to have learnt from and with. I'd like to mention especially Marcus Ahnve, Johannes Brodwall, Enrique Comba Riepenhausen, Andrew Dalke, Greg Dziemidowicz, Dave Hoover, Jon Jagger, Arnulf Krokeide, Robert C. Martin, Dave Nicolette, Thomas Nilsson, Danilo Sato, Christophe Thibaut, Francisco Trindade. Thankyou to all of you.

Some of the material in this book is drawn from the codingdojo.org wiki[2], which is owned by Emmanuel Gaillot. I was one of the many early contributors there, and I am very grateful to everyone who participated in forming that wiki into a useful resource.

Many of the Katas in this book have been designed by other people, and some of the other material as well. I'd like to

[2] http://codingdojo.org

thank everyone who gave me permission to include their Katas in the catalogue, their Dojo Disasters, their wisdom born of experience: Johannes Brodwall, Emmanuel Gaillot, Terry Hughes, Jon Jagger, Robert C. Martin, Roy Osherove, Matt Wynne.

I also want to thank Corey Haines for the work he has done popularizing the Code Retreat, which although different in form, has a philosophy in congruence with the Coding Dojo.

I must also thank my children's violin teachers, especially Marika Wirung and Sven Sjögren. They patiently demonstrate good pedagogy week after week, using the Suzuki method. I have learnt a huge amount about how to teach, by observing them.

I would like to thank all the people who reviewed this book, including Johannes Brodwall, Olivier Demeijer, Nicolas Dermine, Greg Dziemidowicz, Jonas Granqvist, Yves Hannoulle, Jon Jagger, Arnulf Krokeide, Mark Longair, David Read, Anders Schau Knatten, Martin Svalin, Joel Trottier-Hebert, Fredrik Wendt, Joseph Yao. It's a much better book because of your comments.

How to Read This Book

This is supposed to be a practical, useful manual. Dip in and out, or read it all the way through, as you wish. The first section is all about the various coding games and activities you can play with. If you're experienced running Coding Dojos already, you might want to skip most of the second section, which is largely about how to set up and run a new dojo. The third section explains some of the skills you're trying to improve at, and gives you help choosing the right kinds of exercises for your Deliberate Practice. The fourth section, the Kata Catalogue, lists all the Katas I've found useful in the dojo, and you can choose one to tackle at your next meeting. You will be holding a Coding Dojo, right? That's part of the deal with buying this book!

Dojo Disasters

Most of the time we have a really good time in the dojo, and people come away feeling positive about the experience, and what they learnt. Occasionally though, things don't work out so well. In several places dotted about the text you'll find "Dojo Disasters" - little stories where I, and other dojo pioneers, have learnt the hard way.

What is a Coding Dojo?

A Coding Dojo is a meeting where a bunch of coders get together, code, learn, and have fun. It's got to be a winning formula! Programmers generally love the plain activity of writing code, away from managers and deadlines and production bugs. When they've got over their shyness, most are delighted to show others how well they can actually write code, as well as to pick up tips and advice from them. Throw in a suitably puzzling Code Kata and a safe environment to discuss topics like design, testing, refactoring, choice of code editor, tools… and you're away! You'll hardly be able to stop them talking and writing code and learning from one another!

There are few obligatory elements to a coding dojo, designed to promote the aims of learning and having fun. Within those constraints, you still have a lot of freedom to adapt the form and activities according to what you discover suits your group, or in other words, makes it more fun. Some people just prefer to meet with some like minded coders and hack at something together. That's absolutely fine, and can be great fun, but I think you'll learn more if you add just a little more structure.

Essential Dojo Elements

For a dojo I think you need to:

- Hold an intro and retrospective

- Write tests as well as code
- Show your working
- Have moderation or facilitation

The intro and moderation are designed to make sure everyone feels safe to experiment and learn. The retrospective makes sure you reflect on what you've learnt. Writing some tests as well as code sets you up with a feedback mechanism on whether your code is working as you expect. Demonstrating how you write the code, not just the code you end up with, means you learn a mechanism to produce good code, not just what good code looks like. Those elements - intro, retrospective, moderation, showing working, and tests - are what sets a coding dojo apart from any other kind of coding meeting.

The rest of this book explains how a Coding Dojo works in detail.

Section 1: Collaborative Games for Programmers

There are many ways to organize a group of programmers so that they can code and learn together, and in this section I'd like to introduce some of them. There are *whole-group-programming-together* activities, *working-in-pairs* activities, and *look-at-me-coding!* presentations. I like to talk about "collaborative games" for programmers, because that's what we're doing. There are rules, there are activities, there are people talking and helping each other and learning.

What is a Collaborative Game?

A Collaborative Game is one where there is no individual winner, but rather all the participants must contribute to a solution, and you together beat the game itself. I'm a pretty big fan of board games, my cupboard at home is overflowing with strategy games like *Settlers of Catan*, *Seven Wonders*, *Ticket to Ride*, *Dominion*, *Diplomacy*…

None of those titles are a Collaborative Game - in all of them you're competing with the other players, although there is often a degree of collaboration too. Recently I've been discovering I actually quite enjoy playing purely collaborative games, too. For example, *Forbidden Island*, where it's a race against time and tide. The players must work together to gather all the treasures and fly off in a helicopter before the island sinks under the sea. Apart from anything else, when I play it with my children, no individual has to lose, and that makes for fewer tantrums!

I think the coding part of a Dojo should be like a collaborative game, you're not out to appoint a winner, you're there to collaborate and contribute, and solve something together.

Randori

Coding in a group is fun, and this activity takes it to the extreme. Everyone can see the code, projected onto the wall, and everyone gets to write some code, taking it in turns to type. When you get a bunch of half a dozen coders working on the same problem like this, you'll quickly find there are at least a dozen opinions on what code to write! There are some rules designed to keep the Randori on track, and give everyone the best chance to contribute, teach and learn. It can be high volume, intense coding.

A Randori requires almost no preparation, since no-one need have done the kata before. You have to come to design decisions through discussion, and by explaining everything so clearly that whoever has the keyboard can understand what's going on, and decide what direction to take. When you get your turn at the keyboard, suddenly you're in the spotlight, it's hard to think straight, and you have a limited time. You have to choose carefully what code you write - this is your chance to decide exactly what code goes into the codebase, don't waste it!

Before you start, have someone setup their machine, connected to a projector, with an empty failing test. There are a few different variations on exactly where to put the computer, see the next section "Randori Variants". You'll also need to agree who should be the starting pair, and a Pair Switching Strategy.

If the person with the keyboard has an idea for the first test to

write, you could just let the pair get started coding. At some point though, you'll probably want to step back and do some analysis of the problem on a whiteboard. (See the chapter on "States and Moves of TDD", the "Overview" state).

The whole group needs to understand the code that's being written, since everyone will have a turn at the keyboard. Some things are better explained with a sketch on a whiteboard, than by dictating a list of keystrokes to the driver.

In turn, the pair at the keyboard must explain what is going on, so everyone can follow. The audience should give advice and suggest refactorings primarily when all the tests pass. At other times the pair at the keyboard may ask not to be interrupted. See the Randori Rules:

Randori Rules

1. if you have the keyboard, you get to decide what to type
2. if you have the keyboard and you don't know what to type, ask for help
3. if you are asked for help, kindly respond to the best of your ability
4. if you are not asked, but you see an opportunity for improvement or learning, choose an appropriate moment to mention it. This may involve waiting until the next time all the tests pass (for design improvement suggestions) or until the retrospective.

You could appoint a meeting facilitator, who has a special responsibility to see that these rules are followed, but that might not be needed for an experienced group who are

familiar with them. (See also the chapter Facilitating a Dojo Meeting)

Dojo Disaster: Code Ridicule

This dojo disaster story is by Matt Wynne

It was 2008, and I was at an international software conference. I'd only started going to conferences that year, and was feeling as intimidated as I was inspired by the depth of experience in the people I was meeting. It seemed like everyone there had written a book, their own mocking framework, or both.

I found myself in a session on refactoring legacy code. The session used a format that was new to me, and to most of the people in the room: a coding dojo.

Our objective, I think, was to take some very ugly, coupled code, add tests to it, and then refactor it into a better design. We had a room full of experts in TDD, refactoring, and code design. What could possibly go wrong?

One thing I learned in that session is the importance of the "no heckling on red" rule. I watched as Experienced Agile Consultant after Experienced Agile Consultant cracked under the pressure of criticism from the baying crowd. With so many egos in the room, everyone had an opinion about the right way to approach the problem, and nobody was shy of sharing his opinion. It was chaos!

We got almost nowhere. As each pair switched, the code

> lurched back and forth between different ideas for the direction it should take. When my turn came around, I tried to shut out the noise from the room, control my quivering fingers, and focus on what my pair was saying. We worked in small steps, inching towards a goal that was being ridiculed by the crowd as we worked.
>
> The experience taught me how much coding dojo is about collaboration. The rules about when to critique code and when to stay quiet help to keep a coding dojo fun and satisfying, but they teach you bigger lessons about working with each other day to day.

When to choose a Randori form, and what to work on

The Randori approach is most suitable for groups of about 4-10 people. Above that the discussions can get out of hand, and each individual doesn't get much time at the keyboard.

If you choose a Kata that is too difficult, it can be frustrating for the group to get nowhere near finishing it using the Randori form. Particularly at first, try to pick a really simple kata so you can get a sense of achievement from completing it, and having time to make the code really clean.

Pair Switching Strategies

Timebox

- Each pair has a small (5 or 7 minutes) timebox.
- At the end of the timebox, the driver goes back to the audience, the copilot becomes driver and one of the audience step up to be copilot.
- Use a kitchen timer or mobile phone that beeps when time is up.

Note: anecdotally, you need a longer timebox when working in a statically typed language than a dynamically typed one: you have more text to type! Try 7 minutes for Java or C++, 5 minutes for Python or Ruby.

This switching strategy makes it more likely that everyone has a go at driving. The main disadvantage is that you get cut off in the middle of what you're doing, and it can be harder for the next person to pick up where you left off.

Dojo Disaster: Refused Bequest

Kind of like in the Liskov Substitution Principle, if you inherit something you have no use for, it's a sign something is wrong. In the particular dojo I'm thinking of, we had a diverse group where some people had been coding with TDD for many years, and others were young and inexperienced - still at university. We were doing a

> Randori in Pairs, switching pairs every 10 minutes. With only three or four pairs, we got round the table several times. About half way through the kata I went back to a particular machine, and realized I hadn't seen this code before. No, really, it was completely new! The code I had written half an hour previously to pass the current failing test was gone. Vamoosh.
>
> It turns out that one of the less experienced programmers didn't understand my code, so he deleted it. In fact he didn't understand any of the code, and had deleted it all and started again from scratch!
>
> Has that ever happened to you, only with production code? It certainly has to me. We had a great retrospective that time, discussing code readability and reuse.

Ping Pong

1. The driver writes the first test and then hands the keyboard to the copilot
2. The new driver makes the test pass
3. They refactor together, passing the keyboard as necessary.
4. The original driver (who wrote the test) sits down in the audience, and a new person steps up, initially as co-pilot.
5. As step 1, with the new driver (the person who made the last test pass)

This ensures that you don't get broken off in the middle of a sentence like you do with Timebox, and that each person writes both a test and some production code. It has the disadvantage that the pair can spend so long perfecting their code and tests, that not everyone gets a turn at coding. This is particularly likely if there are people present who are unfamiliar with TDD. When they get the keyboard they might not know what to write, and spend a long time before they understand the help they're offered.

NTests

The pair at the keyboard write and implement N tests, where N is usually 1, 2 or 3. Then a different pair steps up to the keyboard. Alternatively only half of the pair is switched after N tests.

I suspect this one only works with pretty experienced TDDers, since you have to be skilled at writing really small tests, and building the solution up gradually. For some coders, this format could tempt them to write too large granularity tests so they can retain the keyboard for longer.

Randori Variants

Driver/Navigator

I've seen it happen many times that an otherwise competent programmer sits down at the keyboard in a Randori and suddenly has no idea what to type. The stress of being in the spotlight causes some kind of biochemical reaction that makes your hands seize up, your mind go blank and your armpits sweat profusely! In this case it can help to separate concerns so the driver is no longer expected to think, only type. Rather like in rally-car racing where the driver drives, and the navigator sits in the passenger seat and tells him or her in detail where to go.

In the Randori, have the non-keyboard wielding half of the pair become the Navigator. This means they do all the thinking, and simply instruct the Driver what code to write. The Navigator can be really specific, even down to the level of "ok, now type 'filter open bracket lambda space x colon…'". Of course most of the time the Driver is actually feeling fairly relaxed, since they only have one thing to worry about: telling the computer what to do. The Navigator can probably just say "filter the list with a lambda expression…". Dictating a sequence of keystrokes is something of a last resort, for when the Driver is having a real rabbit-in-headlights moment!

Once the Driver has been guided by the Navigator for a while, hopefully they'll feel they understand what's going on. When it's time to switch pairs, it could be good to put them into the Navigator role next, and pick a new Driver from the audience.

Co-Pilot stands up

If you're finding the group is not easily able to follow what the pair with the keyboard is up to, you might find it helpful to have the co-pilot, (or navigator), stand up while the driver sits down. This will force them to talk louder. The co-pilot could also stand closer to the projector and point to things on the big screen as they talk. (The driver needs to sit facing the screen in this case, so they can see what's being pointed at).

Facing away from the group

This can be useful if the pair at the front is constantly interrupted, and the discussions often get out of hand. Put a separate table at the front so the coding pair can sit facing away from the group, towards the projector. Without eye contact with the group they will hopefully find it easier to concentrate. It can also be less scary since it's easier to ignore the "audience". It can make it easier for the pair to get going and actually write some code without being pulled in ten different directions by all the backseat drivers.

The main danger with this is of course that the group can get sidetracked and stop paying attention to the code being written.

Section 2: Organizing a Coding Dojo

In the first section we talked about collaborative games you can play while coding in the dojo. What a lot there are to choose from! This section has more practical advice for someone setting up and running a new Coding Dojo. I'll explain how you could structure your meetings, practical details to consider, and talk about the facilitator role. I'd also like to take the chance to explain some theory.

Dojo Theory

The basic premise is that in order to become expert at something, you need to practice. Raw talent, if such a thing exists at all, only gets you so far. Various theories of learning suggest that "Deliberate Practice" over a long period of time is at the heart of attaining expertise.

Deliberate Practice

"When most people practice, they focus on the things they already know how to do. Deliberate practice is different. It entails considerable, specific, and sustained efforts to do something you can't do well—or even at all. Research across domains shows that it is only by working at what you can't do that you turn into the expert you want to become."

– K. Anders Ericsson, Michael J. Prietula, and Edward T. Cokely, writing in the Harvard Business Review

So Deliberate Practice is not the same as reading code or even books about code, valuable as those activities are. As Ron Jeffries points out in his article "Practice: That's What We Do"[3], *"But what changes people is what they do, not what they read. How many diet books have I read? Am I thinner?..."*

[3] http://xprogramming.com/xpmag/jatPractice.htm

Deliberate Practice is not the same as experience gained while doing your job. It is when you actually seek out experiences that will stretch your skills just the right amount, and give you feedback that enables you to learn. I think that it takes a great deal of self-discipline to sit down by yourself and try to do a code Kata, and it can be difficult to get good quality feedback without someone else present or at least available to review your code afterwards.

Going to a Coding Dojo helps enormously because it's fun to socialise and meet other geeks, which means you actually do it, rather than always just intending to sit down of an evening and do a Code Kata instead of watching TV. At the meeting, when you're doing a code kata together, you challenge one another and you have to learn to accept criticism and defend your ideas. You get feedback on not just the code you produce, but your coding technique.

Mastering a skill like Test Driven Development takes a great deal of effort, and it's naive to think you can get all the practice you need while working on production code. Doing all your practice in the dojo is probably ambitious too. I think you'll need to put in some time on your own. If you've enjoyed working on a Kata in the dojo, you might decide you *do* want to switch off the TV for an evening and code it up again instead. You've become motivated by the thought that you can do even better than you did at the dojo, and are looking forward to the next meeting where you can show off what you've learnt.

The dojo should be a good place to meet skilled programmers, and maybe find ones you might like to work with in the future. Some companies sponsor public dojos as a place to recruit programmers for their teams, or to advertise the skills

of their consultants. I see this as a happy side effect though. The real point of going to a dojo is to improve your skills, (and have fun doing so!).

Learning TDD and Downhill Skiing

One of the benefits of emigrating from the UK to Sweden as I have done, is the significant improvement in access to winter sports. I discovered I really enjoy cross-country skiing. It's much like hiking - trekking all day in beautiful terrain, hardly seeing anyone else. This winter, we were in the Norwegian mountains enjoying some cross-country skiing, and for the first time, I decided it might be fun to learn downhill skiing. Mostly so I could keep up with my children, who are keen skiers already! It's quite a different kind of sport - the skis themselves are very different, and of course the slopes are much steeper. While the children were at their ski school one day, I hired a set of skis and boots, and had a go.

The gentle beginner slopes were no problem, I could snowplough just the same as on my cross country skis. I knew this strategy wasn't going to get me far though. If I wanted to go on the steeper slopes and keep up with my daughters, I'd need to master more advanced, parallel turns. A snowplough involves having the skis in a V shape in front of you, and you widen the V on one side to turn in the other direction. For parallel turns, you have to get the skis next to each other, and swing your whole body from side to side as you swish down the slope. It's great fun once you can do it, but while you're learning it's pretty scary. For a fleeting moment while

you're turning, both skis are pointing directly downhill, and you accelerate rapidly!

Still on the gentle beginner slope, I started trying to get my skis next to each other and alter my system of balance and orientation of my body with respect to the slope. It was chaos! Legs and poles and skis in all directions! A slope which I could previously do quite competently with a snowplough, was suddenly really challenging. On several occasions I was grateful for the safety catch that prevented my skis from sliding down the mountain without me.

After some more trial and error I began to get the feel for the new style of skiing, and with almost every run I was able to keep in control at faster speeds. Eventually I was able to tackle a much steeper slope than I would have contemplated on my cross-country skis.

So when you're sitting there doing a code kata using TDD and it feels really awkward, unfamiliar and slow, remember me flailing about on the beginner ski-slope. I know you can probably code a solution to the kata pretty quickly without any tests at all, just like I could ski down that slope with a snowplough. The trouble is, an approach without tests is unlikely to scale to bigger problems. Take some time, suffer some falls, keep writing those tests. With enough practice you'll eventually be coding like a TDD pro, swishing down the mountain with the wind in your hair!

Finding Or Founding A Coding Dojo

When I first experienced the coding dojo, it was such fun I looked around for ways to do it again! At the time there was only one dojo - in Paris - and since I didn't live anywhere near there, it was unfortunately not practical for me to attend. So my approach was to bring Laurent to Sweden to teach me how to do it. I figured that watching someone else doing something is a good way to learn to do it yourself. That probably applies as much to leading a dojo as any coding skills! It worked for me, anyway.

Look around for an existing dojo near where you are. Do some googling, check out meetup.com[4], talk to your friends. If someone has already founded a dojo, but is too busy to run a meeting right now, maybe your offer of help will be all it needs to get it off the ground again! In some cases though, you might find there has never been a coding dojo near where you live.

You might be able to get to a conference where one of the sessions will be a coding dojo. Have a look at conferences like one of the XP series[5] (in Europe), or a conference run by the Agile Alliance[6] (in the US). There might be an "XP Day" or a "Software Craftsmanship" conference, or a "Code Retreat" happening nearer where you are.

[4] http://meetup.com

[5] http://xp2013.org

[6] http://www.agilealliance.org/

If none of that works for you, founding your own dojo could be an excellent move anyway. Even if you've never been to one before, you know how to code, and how to have fun, right? You've also got this book to help you! As a first action, I'd recommend finding someone to co-found it with you. It's more fun that way, and just like with pair programming, you keep each other moving.

In one of the coming chapters I'll go through some of the practical questions you'll have to sort out when you're organizing your dojo. Before launching into that though, I'd like to tell you an encouraging story about a particular Coding Dojo. It's about how a group of enterprising Frenchmen got the whole thing started.

Section 3: Teaching & Learning In the Dojo

What could you learn in your dojo? That's one way of looking at it, but equally importantly - what could you teach? Everyone has different strengths, knowledge, and experience with various languages and tools. In the dojo you ought to meet people you can learn from in some areas, and teach in others. If you know something, being forced to explain it to a beginner can help you understand it even better, so you both teach and learn at the same time!

Skills like pair programming, reading other people's code, writing clean code, automated testing and articulating your ideas are the basis of everything that goes on in the dojo. At some meetings you might want to home in on particular skills or techniques. You might decide to do a kata you know well and have solved lots of times, in order to practice something else. For example, an unfamiliar programming language, editor, IDE, testing framework, or library.

Before I explain about these skills we want to learn in more detail, I'd like to go through the Dojo Principles. I love the way they are so succinct and Zen-like, and remind you that you come to the dojo in order to both teach and learn.

Later in this section I'll be talking a lot about which Code Katas to use while you're learning Test Driven Development, (TDD). This is one of the key skills you're trying to improve at in the dojo. In fact, one of the Dojo Principles says *"code without tests simply doesn't exist."*! I'll also talk about Katas that help you to learn about other styles of TDD, and Functional Programming. The remaining part of the section is a couple of essays I've written about what TDD actually is, and how to write good tests.

Dojo Principles

These principles were written by Christophe Thibaut, and first published in Laurent Bossavit's blog[7] *in 2005, as a guide for new members of the first dojo, in Paris, France. (I have edited them in minor ways to improve readability.)*

The First Rule

One important rule about the Dojo is : At the Dojo one can't discuss a form without code, and one can't show code without tests. It is a design training place, where it is acknowledged that "the code is the design" and that code without tests simply doesn't exist.

Finding a Master

The master can't be a master of every form. I feel quite at ease with recursive functions and list processing e.g. but I think I don't know how to create even a simple web app. Fortunately, while it's the first time they really deal with "tail-recursion" some practitioners here have done professional web apps for years!

Come Without Your Relics

Of course, you know how to do it. You know how and why this code is better than that one. You've done it already. The

[7] http://bossavit.com/dojo/archives/2005_02.html

point is to do it right now, explain it to us, and share what you learned.

Learning Again

In order to learn again something, we just have to forget it. But it's not easy to forget something when you're alone. It's easier when we give our full attention to someone who is trying to learn it for the first time. We can learn from other people's mistakes as well as from our own if we listen carefully.

Slow Down

Learning something should force you to slow down. You can go faster because you learned some tricks, but you cannot go faster and learn at the same time. It's OK, we're not in a hurry. We could do that for years. Some of us certainly will. What kind of deadline will we miss if we spend four more weeks on this code kata rather than on four different katas? More precisely, when we reach the next plateau, is it because we went through the previous one, or is it just because we were flying over it?

Throwing Yourself In

At some point someone will begin to master a particular Code Kata, and want to approach another one. Those threatened by boredom should throw themselves first into a Prepared Kata presentation.

Subjecting To A Master

If it seems difficult to you, look for other practitioners who can judge your code and could easily show something new about it to you. Ask again until the matter contains absolutely no more difficulty to you.

Mastering A Subject

If it seems easy to you, explain it to others who find it difficult. Explain it again as long as they find it difficult.

Section 4: Kata Catalogue

There are many, many code katas, and this catalogue is in no way exhaustive. These are some of my favourites, and ones which I've found to work well in the context of a coding dojo.

What is a Code Kata?

A Code Kata is a small, fun problem that shouldn't take you more than an hour or two to solve in your favourite programming language. The rule is that you must repeat the exercise, and every time try to improve the way you solve the problem. Not just the code you end up with, but the process by which you get to it.

I don't think learning a code kata has anything to do with learning a sequence of keystrokes or perfectly imitating some kind of "master" programmer. That's where the analogy with Karate breaks down! When you "know" a kata, that means that solving the actual problem no longer presents any difficulty to you, and you can concentrate on improving all the other aspects of *how* you solve it. You'll be able to try out a variety of approaches: object oriented, functional

languages, big tests, small tests, another order of tests, with and without faking it, refactoring at this point or that point, different datastructures, algorithms, names… Every time you do the kata, you can try out something new, or make a small improvement to an approach you've used before.

Dojo Disaster: The Architect's Kata

Emmanuel Gaillot recounted for me an incident when somebody new turned up to the Paris dojo. He described himself as a "software architect", and he suggested that not all katas need involve coding. He instead proposed a "design" kata. The group discussed the idea, and the fact that they'd set up the dojo as a place where you learn by **coding** infront of others. On the other hand, someone suggested that in order to really understand a rule, maybe you should break it and see what you can learn.

So they decided to take up the architect's suggestion, and spent an evening drawing boxes and arrows. It didn't turn out so well. As Emmanuel put it: "It was excruciating!". Everyone agreed it was not fun at all. So they kept the rule about coding - in fact, all the Katas in this catalogue involve writing code.

So I agree with Emmanuel, (see the Sidebar "The Architect's Kata"), a code kata must also involve writing actual code. And tests!

How to choose a good Kata for your dojo

The most important thing is to choose a Kata you will enjoy doing! Flip through the catalogue and pick out any topics that look interesting. Have a look at the section "Contexts to use this Kata" for an idea of what you might learn from it. If there is a skill you're working on, there is some advice in the previous section "Teaching & Learning In the Dojo", with suggestions of which Katas are particularly useful.

About this Catalogue

Each Kata has an explanation of the problem to be solved, and links to where you can download starting code (if applicable). In addition, I've added some suggestions to help you get the most out of the kata, and to choose one appropriate for your context.

Additional discussion points for the Retrospective

After you've done the kata, these questions might prompt interesting discussion. (You might be having a great discussion anyway, of course!) When I'm facilitating a dojo, I often find the retrospective is the hardest part. I can see that the group has learnt lots through doing the Kata, but I don't always know how to get people talking about it. That's why I've written these extra notes, to remind me of some questions that might spark good discussion.

Ideas for after the Dojo

If you've done this kata in a dojo, you might be inspired to try it again by yourself at home. Here are some ideas for how to extend the kata or vary it in some way, so you get the most out of it. If several dojo participants continue to work on a kata after the dojo, you can go online to share code snippets, ideas and links, and to continue to discuss what was said in the meeting. Alternatively you could share what you've learnt at the next dojo meeting.

Contexts to use this Kata

If you're in a particular situation, any individual kata might be more or less suitable. This section should help you to choose a good Kata, and help you prepare for your dojo meeting.

Kata: FizzBuzz

Imagine the scene. You are eleven years old, and in the five minutes before the end of the lesson, your Maths teacher decides he should make his class more "fun" by introducing a "game". He explains that he is going to point at each pupil in turn and ask them to say the next number in sequence, starting from one. The "fun" part is that if the number is divisible by three, you instead say "Fizz" and if it is divisible by five you say "Buzz". So now your maths teacher is pointing at all of your classmates in turn, and they happily shout "one!", "two!", "Fizz!", "four!", "Buzz!"… until he very deliberately points at you, fixing you with a steely gaze… time stands still, your mouth dries up, your palms become sweatier and sweatier until you finally manage to croak "Fizz!". Doom is avoided, and the pointing finger moves on.

So of course in order to avoid embarrassment in front of your whole class, you have to get the full list printed out so you know what to say. Your class has about 33 pupils and he might go round three times before the bell rings for breaktime. Next maths lesson is on Thursday. Get coding!

Write a program that prints the numbers from 1 to 100. But for multiples of three print "Fizz" instead of the number and for the multiples of five print "Buzz". For numbers which are multiples of both three and five print "FizzBuzz".

Sample output:

```
1
2
Fizz
4
Buzz
Fizz
7
8
Fizz
Buzz
11
Fizz
13
14
FizzBuzz
16
17
Fizz
19
Buzz
```

... etc up to 100

Additional discussion points for the Retrospective

- Is the code you have written clean? Are there any smells?
- Did you refactor throughout or do it all at the end?
- What if a new requirement came along that multiples of seven were "Whizz"? Could you add that without editing the existing code? (Cue discussion of the Open-Closed Principle)

Ideas for after the Dojo

- When you've got it all working for "Fizz" and "Buzz", add "Whizz" for multiples of seven
- Then add "Fizz" also for all numbers containing a 3 (eg 23, 53)

Contexts to use this Kata

I find this an excellent kata for introducing beginners to TDD. It's pretty straightforward to choose the order of test cases, work in small steps, and complete the whole exercise still leaving time for a decent retrospective.

Kata: Tennis

Tennis has a rather quirky scoring system, and to newcomers it can be a little difficult to keep track of. The Tennis Society has contracted you to build a scoreboard to display the current score during tennis games. The umpire will have a handset with two buttons labelled "player 1 scores" and "player 2 scores", which he or she will press when the respective players score a point. When this happens, a big scoreboard display should update to show the current score. (When you first switch on the scoreboard, both players are assumed to have no points). When one of the players has won, the scoreboard should display which one.

Your task is to write a "TennisGame" class containing the logic which outputs the correct score as a string for display on the scoreboard. You can assume that the umpire pressing the button "player 1 scores" will result in a method "wonPoint("player1")" being called on your class, and similarly wonPoint("player2") for the other button. Afterwards, you will get a call "getScore()" from the scoreboard asking what it should display. This method should return a string with the current score. *(Note: do modify the method names to match the idiom for your programming language)*

You can read more about Tennis scores here[8] which is summarized below:

1. A game is won by the first player to have won at least four points in total and at least two points more than

[8] http://en.wikipedia.org/wiki/Tennis#Scoring

the opponent. The score is then “Win for player1” or “Win for player2”

2. The running score of each game is described in a manner peculiar to tennis: scores from zero to three points are described as “Love”, “Fifteen”, “Thirty”, and “Forty” respectively.
3. If at least three points have been scored by each player, and the scores are equal, the score is “Deuce”.
4. If at least three points have been scored by each side and a player has one more point than his opponent, the score of the game is “Advantage player1” or “Advantage player2”.

The Tennis Society has agreed that Sets and Matches are out of scope, so you only need to report the score for the current game. However, they have requested another feature with lower priority. They want to be able to change the name of the players from “player1” to “Björn Borg” and “player2” to “John McEnroe”. This feature has been categorized “Nice to have”, or, more accurately, “in your dreams”!

Tennis Refactoring Kata

Imagine you work for a consultancy company, and one of your colleagues has been doing some work for the Tennis Society. The contract is for 10 hours billable work, and your colleague has spent 8.5 hours working on it. Unfortunately he has now fallen ill, although he says he has completed the work, and the tests all pass. Your boss has asked you to take over and spend an hour or so on it so she can bill the client for the full 10 hours. She instructs you to tidy up the code a little

and perhaps make some notes so you can give your colleague some feedback on his chosen design.

There are three scenarios for this refactoring kata - imagine three different consultancy companies each with their own solution to the problem. I suggest you start with the first version of the code. When you've got that looking beautiful, start over with the second and third versions.

What is nice about this Kata is that the tests are almost exhaustive, and fast to run, so any mistakes you make while refactoring should be very obvious. You should not need to change the tests, only run them often as you refactor. The code is available on github[9], for several popular programming languages.

I also recommend that if you're doing this as a refactoring kata, that you use a tool to record your session, (see the chapter (#ToolsForTheDojo)), so you can review how large steps you took. The aim is for as small as possible, with as few refactoring mistakes as possible.

Additional discussion points for the Retrospective

- Is the code you have ended up with clean? Are there any smells?
- Are your tests exhaustive?
- Does your code express the scoring rules of Tennis in a readable manner?

[9] https://github.com/emilybache/Tennis-Refactoring-Kata

Refactoring version

- How did it feel to work with such fast, comprehensive tests?
- Did you make mistakes while refactoring that were caught by the tests?
- If you used a tool to record your test runs, review it. Could you have taken smaller steps?
- Did you ever make a refactoring mistake and then back out your changes? How did it feel to throw away code?
- If you never backed out any refactoring mistakes, is that because you're very skilled at refactoring?

Ideas for after the Dojo

- If you did this as a normal kata, try it as a refactoring kata (code on github[10])
- If you've done one of the three refactoring katas, try the other two. Were they easier or harder?
- Try doing all your refactoring without running the tests until you're "finished". How many tests did you break via refactoring mistakes?

Contexts to use this Kata

This is a good kata for practicing refactoring. There aren't many situations where you have the luxury of exhaustive tests. The three refactoring variants have slightly different challenges. The first two are by junior coders with poor grasp of the language. The third is designed to be as concise as possible, to the point of unreadability.

[10] https://github.com/emilybache/Tennis-Refactoring-Kata

Kata: Minesweeper [11]

Have you ever played Minesweeper? It's a cute little game which comes within a certain Operating System whose name we can't really remember. Well, the goal of the game is to find all the mines within an MxN field. To help you, the game shows a number in a square which tells you how many mines there are adjacent to that square. For instance, take the following 4x4 field with 2 mines (which are represented by an * character):

```
* . . .
. . . .
. * . .
. . . .
```

The same field including the hint numbers described above would look like this:

```
* 1 0 0
2 2 1 0
1 * 1 0
1 1 1 0
```

You should write a program that takes input as follows: The input will consist of an arbitrary number of fields. The

[11] This Kata was originally published by the University of Brazil as part of an international contest. http://acm.uva.es/p/v101/10189.html.

first line of each field contains two integers n and m (0 < n,m <= 100) which stands for the number of lines and columns of the field respectively. The next n lines contains exactly m characters and represent the field. Each safe square is represented by an "." character (without the quotes) and each mine square is represented by an "*" character (also without the quotes). The first field line where n = m = 0 represents the end of input and should not be processed.

Your program should produce output as follows: For each field, you must print the following message in a line alone:

```
Field #x:
```

Where x stands for the number of the field (starting from 1). The next n lines should contain the field with the "." characters replaced by the number of adjacent mines to that square. There must be an empty line between field outputs.

This is the acceptance test input:

```
4 4
* . . .
. . . .
. * . .
. . . .
3 5
* * . . .
. . . . .
. * . . .
0 0
```

and output:

```
Field #1:
* 1 0 0
2 2 1 0
1 * 1 0
1 1 1 0

Field #2:
* * 1 0 0
3 3 2 0 0
1 * 1 0 0
```

Additional discussion points for the Retrospective

- What order did you implement test cases in? Was this the best order?
- Does your solution cover all the important edge cases? Really, I do mean *edge* cases!
- What datastructure did you choose to store the minefield in? Would another datastructure be more convenient? What are the tradeoffs? Would a different choice affect which test cases you should write?

Ideas for after the Dojo

Implement KataMinesweeper again using a different datastructure to store the minefield in. Alternatively, try the Kata Game of Life with the same datastructure as you used in Minesweeper.

Further Reading

If you've enjoyed this book, and are finding it useful in your Coding Dojo, you might also like to read some of these books. Many of them contain worked code examples that you could go through in the dojo, and perhaps turn into Code Katas. Some of them you'll find I already have done! In any case, they're books that you have to do more than just *read* to get the most out of. They're full of *code*, and you're a *coder*, right?

Refactoring and Design

- "Refactoring: Improving the design of existing code", Martin Fowler
- "Refactoring to Patterns", Joshua Kerievsky
- "Working Effectively with Legacy Code", Michael Feathers
- "Agile Software Development: Principles, Patterns and Practices", Robert C. Martin

TDD, Clean Code

- "Test-Driven Development by Example", Kent Beck
- "The art of Unit Testing with examples in .NET", Roy Osherove
- "Clean Code", Robert C. Martin
- "Code Complete", Steve McConnell
- "xUnit Test Patterns", Gerard Meszaros

London School of TDD

- "Growing Object Oriented Software, Guided by Tests", Steve Freeman and Nat Price
- "The RSpec book", David Chelimsky et al

Functional Programming

- "Functional Programming for the Object-Oriented Programmer", Brian Marick

Interesting Books for Coders (except with less actual code)

- "Extreme Programming Explained", Kent Beck
- "The Pragmatic Programmer", Andrew Hunt and David Thomas
- "Apprenticeship Patterns: Guidance for the Aspiring Software Craftsman", Dave Hoover, Adewale Oshineye

A sequel to this book

To expand on some of the topics in this book, I'm writing a sequel, which you might be interested in. It uses Code Katas to illustrate coding techniques:

- "Mocks, Fakes and Stubs"[12]

[12] https://leanpub.com/mocks-fakes-stubs