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

ISBN 978-91-981180-0-1

©2012 - 2013 Emily Bache. Cover picture copyright Topaz/F1online.

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.