Coding Dojo: an environment for learning and sharing Agile practices

Danilo Sato
ThoughtWorks Limited
dsato@thoughtworks.com

Hugo Corbucci, Mariana Bravo
Department of Computer Science
University of São Paulo, Brazil
{corbucci, marivb}@ime.usp.br

Abstract

A Coding Dojo is a meeting where a group of programmers gets together to learn, practice, and share experiences. This report describes the author's experience of creating and running and active Coding Dojo in São Paulo, Brazil, sharing the lessons learned from the experience. The role of the Dojo in the learning process is discussed, showing how it creates an environment for fostering and sharing Agile practices such as Test-Driven Development, Refactoring, Pair Programming, among others.

1 Introduction

> In software we do our practicing on the job, and that's why we make mistakes on the job. We need to find ways of splitting the practice from the profession. We need practice sessions.
>
> –Dave Thomas

The idea of a *Code Kata* was initially proposed by Dave Thomas as an exercise where programmers could write throwaway code to practice their craft outside of a working environment [9]. Laurent Bossavit later proposed the idea of a *Coding Dojo*: a session where a group of programmers would gather to solve the *Code Kata* together [3]. Although the session is organized around a programming challenge, the main goal of a *Coding Dojo* is to learn from others and improve design and coding skills through deliberate practice. This creates a learning environment where Agile technical practices, such as those proposed by Extreme Programming (XP) [2], can be shared.

This report describes the authors' experience of founding and running a *Coding Dojo* in São Paulo, Brazil. Section 2 will present the concept and rules of a *Coding Dojo* and the tailored process to conduct the sessions in São Paulo, improved over time by retrospectives. Section 3 will present lessons learned from the weekly meetings being held since the first session in July, 2007. Section 4 will discuss the aspects of a *Coding Dojo* that foster learning and tacit knowledge sharing, concluding in Section 5.

2 Coding Dojo

A *Coding Dojo* is a periodic meeting (usually weekly) organized around a programming challenge where people are encouraged to participate and share their coding skills with the audience while solving the problem. The main principles of the *Coding Dojo* are to create a **Safe Environment** which is collaborative, inclusive, and non-competitive where people can be **Continuously Learning**. Some of the XP principles align nicely with that [2], such as **Failure** – it is OK to fail when learning something new – **Redundancy** – one can always gain new insights when tackling the same problem with different strategies – and **Baby Steps** – each step towards the solution should be small enough so that everybody can comprehend and replicate it later.

There are some general rules that allow the *Coding Dojo* session to be productive and to flow. The meeting is held in a room with enough space for all the participants and usually requires only a projector and a computer or laptop. Having whiteboard space for sketching and design discussions is also valuable. The participants are encouraged to develop the solution using Test-Driven Development (TDD) [1] and are free to choose whichever programming language they prefer. There are two main meeting formats:

- **Prepared Kata**: In this format, someone has already solved the proposed *Kata* prior to the meeting (alone or in group) and the solution is presented to the audience during the session. Instead of showing the final code and tests, the presenters start from scratch, explaining each step and allowing the other participants to ask questions or make suggestions. The session goal is that everyone should be able to reproduce the steps and solve the same problem after the meeting.

- **Randori**: In this format, the participants solve the problem together, following TDD and Pair Programming in time-boxed rounds (usually between 5 and 7 minutes). At the end of each turn, the pilot joins the audience, the co-pilot becomes pilot, and a new co-pilot joins the pair from the audience. An extra rule is that discussions and suggestions should be only given when the pair arrives in a green bar, with all the current tests passing. The reason is that, while on a red bar, the pair should focus and work together to get the tests passing. The audience can always suggest refactorings and optimizations on a green bar.

These formats allow the creation of an environment where participants can discuss and practice a wide range of topics, such as: TDD, Behaviour-Driven Development, Agile, Refactoring, Pair Programming, Object-Oriented Design, Algorithms, different programming languages, paradigms, and frameworks.

2.1 Coding Dojo@SP: History and Process

The meetings in the São Paulo *Coding Dojo* started in 12^{th} of July, 2007 and have been held weekly since then in the Institute of Mathematics and Statistics of the University of São Paulo. Some extra sessions were done during the University holidays and most of the session reports are available on the international *Coding Dojo* wiki[5]. The number of participants varied from 3 to 16 and their skill level ranged from undergraduate students to experienced programmers.

On the first meeting the participants were asked to fill index cards with their expectations and personal interests in attending the sessions. An affinity map was built with that information and the three main interests were to practice problem solving skills, to learn different ways and algorithms to solve the challenges, and to learn new programming languages. Some of the sessions were highly focused on design problems and algorithms, which left less time for writing code, but the participants liked to learn from the discussions. On the other hand, the majority of the sessions required less design and algorithms discussion, leaving more time to write code and allowing the participants to experiment with a wide range of programming languages, such as Java, C, Ruby, Python, Lua, and Smalltalk.

The sessions usually follow the same process:

- **Problem Choosing**: Before the meeting, the participants receive an e-mail with 3 to 5 options of problems to be solved. The problems are chosen from several sources (such as Ruby Quiz[1], Programming Challenges[2], UVa[3], and SPOJ[4]). Each option is briefly presented and the participants vote on which problem will be solved in the meeting. This usually takes 5 to 10 minutes.

- **Problem Discussion**: Once the problem is chosen, the group discusses the different approaches to solving it, usually ending up with an agreed approach and a list of TO-DO items, as proposed by Kent Beck [1], to guide the pairs during the implementation. This usually takes 10 to 20 minutes, but there were meetings when the group spent the entire meeting discussing algorithms and possible approaches to a complex problem.

- **Coding Session**: With an agreed approach to solve the problem, the participants start the coding session in one of the two formats – a *Prepared Kata* or a *Randori*. They should practice Pair Programming and Test-Driven Development as a general rule. This usually takes 1 to 2 hours.

- **Retrospective**: At the final 10 to 20 minutes of the session, the participants stops coding (even if the problem was not completely solved) to reflect on the experience and share the learned lessons with the group. This is also a good time to discuss what could be improved and to come up with action items for the next meeting.

Finally, the São Paulo *Coding Dojo* came up with two special roles that can be rotated between participants, but that are very important to organize and to make sure the meetings continues to happen. The **Moderator** or **Organizer** is responsible for what happens before, during, and after the meeting. He handles tasks such as reserving the meeting room, sending reminders and options of problems to be solved, setting up the computer and projector prior to the meeting, moderating discussions, conducting the retrospective, and cleaning up the room after the session. The **Scribe** is responsible for publishing the results of the session and sharing it with the people that could not attend the meeting. He handles tasks such as posting the session report to the wiki, publishing the final source code to the group, sometimes taking photos, and documenting the results of the retrospective.

3 Lessons Learned

After over 6 months of weekly meetings, the authors have identified aspects of the Dojo that went well, aspects that went less well and aspects that still are puzzles to understand.

[1] http://www.rubyquiz.com/

[2] http://www.programming-challenges.com/

[3] http://acm.uva.es/p/

[4] http://www.spoj.pl/

Since the sessions are being held, the authors could identify what practices and rules went well (Subsection 3.1) but also found out that some things work less well (3.2) than expected. Finally, applying the practice to different audiences and in different contexts, the authors discovered some unaddressed issues (3.3).

3.1 What went well?

3.1.1 Retrospectives and action items

As described in the previous section, every meeting is ended with a short retrospective. The participants receive red and yellow sticky cards and write positive and negative aspects of the session that just ended. In the begining, the group followed a simple retrospective format, asking "What worked well?" for positive aspects and "What can be improved?" for negative aspects. These questions led people to write items about the process used for the meetings, such as when to choose the problem, when to change the programming language, what laptop to use, etc. This kind of feedback helped improve the Sao Paulo *Coding Dojo* itself, and to reach the process described in 2.1.

After some time, the retrospective format changed to reflect the objetives of the *Coding Dojo*. Now participantes are asked to think about the following questions:

- **"What have we learned?"**: Reflecting and discussing what was learned is an effective way to make learning an active process and to verify that the session met its goals.
- **"What has hindered learning?"**: The negative aspects of a meeting are discussed, and the main impediments are identified. For these impediments, the group thinks of what could be done to eliminate them, and the action items are recorded for future meetings.

3.1.2 The goal is not to finish

When the Sao Paulo *Codingo Dojo* started, the participants agreed that one of the goals was to learn different algorithms and approaches to problem solving. On the second meeting, during the *Randori* coding session, the time-boxed rounds became a race of who could produce more code and get closer to solving the problem. The coding happened really fast and soon some participants could not keep up with what changes were made and why.

At the following retrospective, the group decided that finishing the problem should never be a goal of the meeting. More than that, it was agreed that not writing the entire solution was OK, as long as the participants could learn something from the coding session. "It's OK not to finish" has been one of the Sao Paulo *Coding Dojo*'s principles ever since, and it is repeated whenever someone forgets or doesn't know.

With that principle clear, the participants take their time writing code and undestanding it, and the group often does not finish implementing entire solutions to the problems.

3.1.3 Time-boxing

The Sao Paulo *Coding Dojo* has always used 7 minutes time-boxes for *Randori* sessions. However, for a long time the group disrespected a bit the time-boxes. That is, if a pair was in the middle of writing a piece of test or a refactoring, and the group considered this activity to be short, the pair was allowed to finish the current code before switching. At first this took 1 or 2 minutes more, but this overtime gradually increased until there was no more a time-box, but a minimum time for each pair.

This actually made it difficult for everyone to be focused on the big screen - the longer a pair stayed at the front, less and less people payed attention to them. As a result, the group decided to adopt *really strict* time-boxes. When the timer rings, no matter what else, the pair is switched. This has made meetings more dynamic and easy to follow.

One side-effect of this approach is that if some discussion happens between the group, the current pair has less coding time. The participants have not yet found a solution to this, but some ideas have been suggested and should be attempted at future meetings.

3.1.4 Information radiators

Since the *Coding Dojo* uses TDD, the coding session follows a clear cycle: red - green - refactor. However, between discussions or distractions, the group sometimes forgets what is the current stage. The participants felt the need of visual feedback of the current stage - such as an information radiator. Therefore, some means of displaying information have been used and tested.

The first was a red/green window developed individually by one of the participants. The program would collect test results pushed to a temporary file and display a color appropriatly. Although a bit buggy, it proved an important tool for displaying information about the TDD cycle. When the programming language used in the sessions switched to Ruby, the group started using autotest, which is a program that watches for changes in the program files and automatically runs related tests. Then, another participant adapted a script to have the autotest results be reported in the OS notifications system, with a little pop up on the top right corner of the screen. When the pop up is red, it stays on the screen until it is clicked.

3.1.5 Inspiration for the meeting

3.2 What went less well?

3.2.1 Moderating Brazilians (hard not to speak on red)

One of the rules imported from international Dojos is that, during a Randori, the audience should not speak when the tests are red. Red time is when the current pair is supposed to practice and make the tests pass, and only if they ask for help should the other participants give suggestions.

However, from the beginning this has been a hard practice to follow at the Sao Paulo Coding Dojo.

One of the problems the Dojo participants have faced from the beginning is that people talk at bad moments. The authors believe this is related to cultural aspects of the group. Brazilian people are very communicative and even if they do not talk to the current programming pair, the other small chats are enough to disturb the order. We tried to fight that and it got a bit better with time but it is no longer a rule. More likely a good practice that attendees try to follow and self control themselves when getting out of hand.

3.2.2 TDD/BDD and algorithms

A few sessions took programming problems from sources in which traditional algorithms such as Dijkstra's shortest path between two nodes on a graph are ment to be implemented to solve the problem. Those experiences points out to be very disapointing in the meaning that, even if all attendees understood the solution, they never got to the solution in the given time. Moreover, the tests or behaviours that were written would hardly direct the group to the desired solution and when they did, refactoring the existing code to the desired solution would be a huge work.

Not only was it frustating not to reach the correct implementation but it was even worse when refactoring for a long period of time without tests running. It broke the **Baby Steps** principle, made it harder to change pairs and lost the focus of the tests. The reason might be that traditional algorithms hardly work partially and can be improved upon.

3.2.3 Balancing randoris and prepared katas

Randori sessions are very important because they provide learning and pratice to all participants. *Prepared Katas* are also interesting since it is usually possible for the group to advance further in seeing the implementation of a problem. However, it takes someone's time outside of *Coding Dojo* sessions for a *prepared Kata* to be developed and practiced. Because of that, this kind of session is much more rare than *Randori* sessions. Although sometimes the group feels the need or opportunity for a *prepared Kata* session, it is not easy to find a participant with availability to prepare it.

3.2.4 Programming environment

Open source communities know the issue very well: Emacs or VI?

Each programmer has her prefered tools, environments, key sets and shortcuts. With the laptop era, the problem applies also to hardware: each laptop according to its origin or manufacturer has a slightly different keyboard. Gathering several programmers that work on different operating systems, software and languages causes a lot issues. Having Apple laptops running Mac OS X with Command keys instead of Control brought several complains from attendees. Attempts to change the environment brought the same issues with other people.

Finding an environment less hostile to attendees is still a problem for a meeting that hopes to bring all sorts of people. So far the issue has been addressed by trying to stick to the same environment so that people get used to it.

3.3 What is still puzzling?

After over 50 *Coding Dojo* sessions, lots of issues were found and solved. However, some of them still puzzle the authors. Those questions are an invitation to discussions and attempts.

3.3.1 How to reach a wider audience?

As discussed in Section 4, the coding sessions are very effective spreading knowledge among attendees. Knowledge in a *Coding Dojo* session is similar to value in open source software: it grows as more people add their own knowledge in. It is therefore natural to have a will to bring more and more people to those session. But, even in free software, people do not throw in their knowledge if there are no compensation to it ([8]) so the session must bring knowledge to every attendee.

The authors found out that gathering more than a certain amount of individuals (for example 20) in a *Coding Dojo* rises serious problems. The gap of knowledge tends to be greater which can lead to intimidation of certain attendees and lack of interest from others. It also gives the impression of having a slower dynamic since it is always someone else coding than the attendee himself. Lastly, it increases the temptation to talk to others attendees in the audience. The result is that people do not agree on an implementation and keep erasing what the previous pair did. Knowledge is then never shared and the session looses its meaning.

Is it possible to fight those problem? Can one session hold many people and still spread enough knowledge to each attendee to have them benefit from the meeting? If not, should the meeting be split? How to balance attendees to have them benefit from it?

3.3.2 How to share efforts with the community?

Following the same motivation previously presented, if it is possible to share results between *Coding Dojos*, it would bring even more value to those communities. Results can mean code, software or even practices and sharing them should allow other communities to go a step further. Again, it relates very closely to the whole open source idea but differs on the media that should disseminate knowledge. While free software communities have code being the final way to transmit knowledge, the code generated on a *Coding Dojo* session rarely transmits a decent portion of what was learned to achieve that result.

What could help transmit that experience? What software are lacking to improve a *Coding Dojo* session? Could code between very different communities be reused? Are two *Coding Dojo* groups similar enough to have share the same issues?

3.3.3 How to keep attendees engaged?

Since the *Coding Dojo* sessions should evolve with time as attendees get more used to TDD and the language or environment used, it is interesting to keep a participant base that could ensure the normal flow of the session. What makes attendees come back to another session? How to ensure that those characteristics are always present? How handle this goal with the need for new attendees to bring new ideas?

4 Dojo and Learning

The main goal of a *Coding Dojo* is learning through practice. Like a pianist plays scales and a martial arts student practices basic moves, the *Code Katas* serve as focused exercises that allow the participants to improve on specific skills. Ericsson et al. studied what influences the acquisition of expertise in different domains such as music, chess, and sports [7]. They found that deliberate practice over a long period of time (usually more than 10 years) is at the heart of attaining expertise. Their empirical study shows that experts carefully schedule deliberate practice and limit its duration to avoid exhaustion and burnout. Although it takes time to become an expert, the role of deliberate practice is still important through the learning process.

The Dreyfus Model of skill acquisition defines five developmental stages when learning a new skill: novice, competence, proficiency, expertise, and mastery [6]. A *novice* needs a set of pre-defined rules that he can apply to situations without previous experience on the domain. *Competence* comes with experience, when the student can identify recurring patterns and understand his environment. With increased practice and experience, a *proficient* student starts to question the guidelines and is able to apply different rules considering longer term consequences. Once the repertoire of experienced situations becomes so vast, an *expert* student is able to intuitively trigger the appropriate action for a specific situation. According to the Dreyfus model, there is no higher level of mental capacity than expertise, but there are moments when an expert can cease to pay conscious attention to his performance and still produce the appropriate perspective and its associated action, reaching a state of *mastery*.

Although the *Coding Dojo* can not provide the intuition and unconscious competence required to achieve expertise and mastery, the deliberate practice of Agile practices and coding skills can help participants to go from novice to proficient. Also, since there is no single master for every subject, participants of different levels can share their knowledge and improve the learning experience of the whole group.

4.1 Dojo at the University

One of the attendees joined the São Paulo *Coding Dojo* since the first sessions, when he had just finished his first semester in Computer Science. He is now on his third semester and most of his assignments are done using TDD no matter what language is being used. His latest work involved implementing sparse matrices with common operations in C. He decided by himself to implement it using TDD and a simple testing library developed during a *Coding Dojo* session [4]. He was able to write clear code with full test coverage. His ability to identify and pin down the required tests to drive the correct implementation far surpasses his colleagues'.

He has been showing strong evidences that the knowledge and practices obtained from the *Coding Dojo* can be absorbed and understood regardless of prior experience on the subject. Since such testing practices are not part of the regular class' program, it shows how the participation on the *Coding Dojo* can help a novice to become competent. Practices that were just followed as rules in the initial sessions became more natural and could be applied to different contexts and situations. It also shows that the informal, non-directed and non-rigid learning experience can be effective and complement more traditional teaching methods.

4.2 Dojo at ThoughtWorks

More recently, one of the authors had the experience of running a *Coding Dojo* in a different environment: inside a company. He took over the responsibility of running a bi-weekly meeting called "Ruby Tuesdays". The session's goal was to share knowledge between expert and novice developers in regards to the Ruby programming language. Although the focus was on a specific programming language,

when the author became the moderator, he made a presentation and suggested the use of a *Coding Dojo* format for the meetings.

So far the results are very positive. The use of a more structured format allowed the session to flow better and the use of a single projector proved to help everyone follow the same train of thought. The retrospective at the end is also helpful to consolidate the lessons learned and to discuss what can be improved for the next meeting. Running an internal *Coding Dojo* within a company can help developers to share their interests in particular concepts and practices, allowing the rest of the organization to experience the benefits of applying different techniques. It also creates a safe environment, free of normal project pressure, allowing them to conduct controlled experiments before applying the practices on their day-to-day work.

5 Conclusion

This report shares the experiences of running a *Coding Dojo* at the University of São Paulo and, more recently, at ThoughtWorks. The process and roles used to conduct the meetings were improved through retrospectives based on the participants' feedback. By sharing the lessons learned from this experience, the authors expect that this learning tool can be applied to different contexts, encouraging more people to start their own *Coding Dojos*. Finally, the role of a *Coding Dojo* in the learning process was discussed, showing how students at different skill levels can use deliberate practice to improve and to share knowledge with a wider group.

References

[1] K. Beck. *Test Driven Development: By Example*. Addison-Wesley Professional, 2003.

[2] K. Beck and C. Andres. *Extreme Programming Explained : Embrace Change*. Addison-Wesley Professional, 2nd edition, 2004.

[3] L. Bossavit. Object dojo. `www.bossavit.com/pivot/pivot/entry.php?id=207`, 2003.

[4] Coding Dojo São Paulo - Session 31: A short C Unit testing library. `dojo_sp.googlegroups.com/web/31-CTEST.zip`, 2008.

[5] Coding dojo wiki. `www.codingdojo.org`, 2007.

[6] S. E. Dreyfus and H. L. Dreyfus. A five-stage model of the mental activities involved in directed skill acquisition. Technical report, California University of Berkeley Operations Research Center, 1980.

[7] K. A. Ericsson, R. T. Krampe, and C. Tesch-Römer. The role of deliberate practice in the acquisition of expert performance. *Psychological Review*, 100(3):363–406, 1993.

[8] R. A. Ghosh. Cooking pot markets: an economic model for the trade in free goods and services on the internet. *First Monday*, 3(3), 1998.

[9] D. Thomas. Code kata: How to become a better developer. `codekata.pragprog.com`, 2003.