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 authors' experience of creating and running an 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 and 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 [10]. 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 only be 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 pass the tests. The audience can always suggest refactorings and optimizations on a green bar.

Figure 1. A typical Dojo Randori session

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 of the São Paulo *Coding Dojo* started on the 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.

At 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** (5 to 10 minutes): 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.
- **Problem Discussion** (10 to 20 minutes): 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.
- **Coding Session** (1 to 2 hours): 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.
- **Retrospective** (10 to 20 minutes): At the end of the session, the participants stop coding (even if the problem was not completely solved) to reflect on the experience and share the learned lessons with the group.

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 continue 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.

[1] http://www.rubyquiz.com/
[2] http://www.programming-challenges.com/
[3] http://acm.uva.es/p/
[4] http://www.spoj.pl/

3 Lessons Learned

After over 6 months of weekly meetings, the retrospectives allowed the process to be improved. Section 3.1 will discuss the aspects of the *Coding Dojo* that went well. Section 3.2 will discuss the things that worked less well than expected. Finally, in Section 3.3 the experience of running the sessions in different contexts and to different audiences raised some questions for further discussion.

3.1 What Went Well?

3.1.1 Retrospectives and Action Items

As described in the previous section, every meeting ends with a short retrospective [9]. The participants receive red and yellow sticky notes and write positive and negative aspects of the session. In the beginning, the group followed the usual retrospective format, asking "What went well?" and "What could be improved?". 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 São Paulo *Coding Dojo* itself, reaching the process described in Section 2.1.

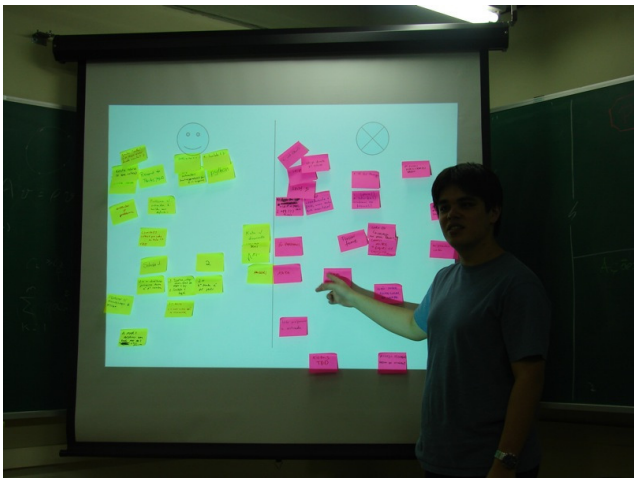

Figure 2. Conducting the retrospective

After some time, the retrospective format changed to reflect the objectives of the *Coding Dojo*. Now participants 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. The group performs a root cause analysis and discusses how these impediments could be eliminated, coming up with a series of action items. People take responsibility to handle each action item for the next meeting, the results are recorded for future reflection, and the effects of the change are re-evaluated in the next retrospective.

3.1.2 The Goal is not to Finish

When the São Paulo *Codingo Dojo* started, the participants agreed that one of the goals was to learn different algorithms and approaches to problem solving. At 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 the goal of a 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 São Paulo *Coding Dojo*'s principles ever since, and it is repeated whenever someone joins the group or simply forgets.

With that principle in mind, the participants take time to write clean and understandable code, and the group often does not finish implementing the entire solution to the problems. Unlike a programming challenge or contest, going fast in a *Coding Dojo* session is not beneficial.

3.1.3 Time-boxing

The São 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 test or finishing 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 picture – 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 they are doing, the pair has to be switched. This allowed the meetings to be more dynamic and easier 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 suggested the option of having a pause requested

by anyone in the audience for a time-boxed discussion. During that time, the coding pair should conduct the discussion and the timer is paused. Another alternative would be to include these time-boxed discussions between each pair change, skipping it if the group feels it is not needed.

3.1.4 Information Radiators

Since the *Coding Dojo* uses TDD, the coding session follows a clear "red - green - refactor" cycle. 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 on an XP team. 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 Perl script would collect test results pushed to a temporary file and display a color appropriately. It proved to be an important tool for displaying information about the TDD cycle and to help the moderator during the coding session. When the programming language was switched to Ruby, the group started using autotest[5], which is a program that watches for changes in the program files and automatically executes the appropriate tests. Then, another participant adapted a script to report the autotest results 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

One interesting tool that was appreciated by the participants was to have an "Inspiration Moment" prior to the discussions. The moderator chooses a card from the Creative Whack Pack [11]. Each card contained a different creativity strategy that inspired the thought process and gave insights into how to approach the problem at hand. Some examples of cards that gave insights during the discussions were to **Ask "Why?"** and **Ask "What if?"**. After hearing the small story contained in each card, the participants were more willing to ask the same questions during the meeting. Other strategies were also valuable when trying to come up with new design ideas, such as **Don't Fall in Love with Ideas**, **Simplify**, and **Reverse**.

3.2 What Went Less Well?

3.2.1 Moderating Brazilians

As described in Section 2, one of the rules of a *Randori* session is that the audience should not speak when the tests are failing. A red bar is the time when the current pair is supposed to practice and get to a green bar. Unless the pair asks for help, the other participants should not give suggestions.

However, from the beginning this has been a hard rule to follow at the São Paulo *Coding Dojo*. One of the problems the participants have faced 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, they like to chat with other peers, disturbing the focus of the session. The moderators tried to fight that and it got a bit better with time but it is no longer a rule. It became more like a good practice that the moderator should remind the participants when things start getting out of control.

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 the appropriate solution. At times, these sessions turned out to be a little disappointing: even if all attendees understood the solution, they never got to the full implementation in the given time. Moreover, the participants found that implementing the simplest solution to make the current test pass, and drive the algorithm through examples of the expected outcome usually required a huge breakthrough refactoring step that maybe could not be found if nobody had the previous knowledge of how the algorithm works. This step usually took several turns to be understood and implemented, during which the tests remained red. It broke the **Baby Steps** principle and gave a feeling that the group was not making progress.

The reason might be that complex algorithms usually require a broader knowledge experience and, unless the programmers have the steps to drive the proper implementation in their heads, they will hardly get to a solution by simply following TDD with examples of input and output pairs.

3.2.3 Balancing Randoris and Prepared Katas

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

[5] `http://www.zenspider.com/ZSS/Products/ZenTest`

3.2.4 Programming Environment

Open source communities know the issue very well: Emacs or VI? Each programmer has his preferred tools, environments, key sets, and shortcuts. With laptops, the problem is also extended 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 to other people.

Finding an environment less hostile to attendees is still a problem for a meeting that hopes to engage 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?

3.3.1 How to Reach a Wider Audience?

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

The authors found that gathering more than a certain number of individuals (about 20) in a *Coding Dojo* raises serious problems. The knowledge gap tends to be greater, leading to intimidation of certain attendees and lack of interest from others. It also gives the impression of having a slower dynamic since it takes more turns for the attendee to code. Lastly, people feel more compelled to talk to other 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 not shared, progress is not made, and the session looses its meaning.

Is it possible to fight these problems? 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 the attendees' skills to have them benefit from the split?

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 be code, software, or even practices and sharing them should allow other communities to go a step further. The code generated on a *Coding Dojo* session rarely transmits the lessons the participants experienced during the meeting to achieve that result. What could help transmit that experience? What tools are lacking to improve a *Coding Dojo* session? Do other *Coding Dojo* share the same issues? If so, have they addressed this and how?

3.3.3 How to Keep Attendees Engaged?

Since the *Coding Dojo* sessions should evolve with time as attendees get more used to TDD, the language, and the environment used, it is interesting to keep a subset of participants that can ensure the normal flow of the session. What makes attendees come back to another session? How do you ensure that these characteristics are always present? How do you balance this goal with the desire to have new attendees bringing 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 can be applied 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 stage of *mastery*.

Although the *Coding Dojo* can not provide the intuition and unconscious competence required to achieve expertise and mastery, deliberate practice can help participants to go from novice to proficient. Also, since there is no single master for all subjects, participants of different levels can share their knowledge and improve the group as a whole.

4.1 Dojo at the University

Running the *Coding Dojo* at the University gave the authors an example of the good impacts of the sessions in one particular student. One of the attendees joined the *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 he uses TDD in most of his assignments, no matter what language is being used. His latest work involved implementing common sparse matrices operations in C. He decided 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 started 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 about the Ruby programming language between expert and novice developers. Although focused on a specific 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 points for improvement. Running a *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 over time 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 share knowledge with a wider group.

Acknowledgements

The authors would like to thank Prof. Dr. Alfredo Goldman and Prof. Dr. Fabio Kon for supporting the São Paulo *Coding Dojo* sessions. A special thanks goes also to Fabricio de Sousa for helping the organization of several sessions. The authors are also very grateful for learning and sharing experiences with all the *Coding Dojo* participants.

References

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

[2] K. Beck and C. Andres. *Extreme Programming Explained : Embrace Change*. Addison-Wesley, 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. `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] N. L. Kerth. *Project Retrospectives: A Handbook for Team Reviews*. Dorset House Publishing Company, 2001.

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

[11] R. von Oech. Creative whack pack, 1989.