The Coding Dojo Handbook

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

Emily Bache

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 2012-11-03

Tweet This Book!

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

The suggested hashtag for this book is #codingdojo.

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

https://twitter.com/search/#codingdojo

Contents

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 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. I'd like to thank everyone who gave me permission to include their Katas in the catalalogue: Terry Hughes, Jon Jagger.

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 would like to thank all the people who reviewed early versions of this book, including Greg Dziemidowicz, Jonas Granqvist, Yves Hannoulle, Jon Jagger, Arnulf Krokeide, David Read. It's a much better book because of your comments.

By publishing this work-in-progress on LeanPub, I hope to receive feedback in the form of comments from the wider community. If you are kind enough to do so, I will be very happy to write your name here in the finished book.

[2]http://codingdojo.org

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. If you're experienced running Coding Dojos already, you might want to ignore most of the text and just skip to the Kata Catalogue to find some good ideas for a Kata to tackle at your next meeting.

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.

Section 1: About the Coding Dojo

What happens at 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 have a meeting where they get together 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.

For a dojo I think you need to hold an intro and retrospective, to write tests as well as code, and have someone moderate discussions. The intro and moderator 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. Those elements - intro, retrospective, moderator and tests - are what sets a coding dojo apart from any other kind of coding meeting.

A Meeting Outline

Doing some planning in advance of your dojo meeting can help you sort out in your head what you expect to happen. It should also help you explain to people who you want to sign up for the meeting. When you make your plan, think about these elements, (you don't necessarily do them in this order):

- Introduce the Dojo
- Agree the activities for today's Dojo
- Code!
- Retrospective

There is a fuller description of each in the next section, with suggestions of how long to spend on each activity.

Introduce the Dojo

This doesn't have to be anything long or fancy, the purpose is to help everyone to feel safe, especially newcomers. If you've never coded in a group before, or you're new to the language and tools being used, it can be rather intimidating at first. You want to try to get everyone into a position where they're ready to take part in the dojo. Perhaps mention the Dojo Principles, go through the Dojo Rules, or simply remind people to be respectful towards one another.

This part of the meeting could take from 2 - 15 minutes depending on how many newcomers there are and how much they want to know about the theory behind the coding dojo.

Agree the activities for today's Dojo

In a later chapter we'll look at some of the specific activities you could try out. (See Meeting Activities). It's a good idea to get some consensus about what the group wants to focus on, and talk a little about what you intend to try to learn. If there is something you want to change compared with what happened at the last meeting, this is the time to mention it.

Again, this section of the meeting should be rather short, 2 - 10 minutes perhaps.

Code!

This is the main part of the meeting, (ie the most fun!). We're all coders after all, and we're here to improve our coding skills. Along with the coding, there should be plenty of discussion, questioning and helpful suggestions. Reminding people of the Dojo Rules (see next section) can help to keep these discussions on track.

Retrospective

This is the part of the meeting where we reflect on what we have learnt in the Dojo. You could use any retrospective form or activity that you think would be helpful. For example you could ask each person to first write down something they learnt, something that surprised them and something they still don't understand, then go around each person asking them to share. The simplest way to do it is just a free discussion about what happened and what people thought about it.

You should be sure to reserve some time for the retrospective so it's not squeezed out by too much enthusiastic coding. Reflection is essential for learning. I suggest 5 - 15 minutes.

Dojo Theory and Background

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 (particularly those proposed by Dr K. Anders Ericsson) suggest that "Deliberate Practice" over a long period of time is at the heart of attaining expertise.

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?...*"

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 CodingDojo helps enormously because it's fun to go and 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 practice, and it's naive to think you can get all the practice you need either on the job or in even the dojo. I think you'll need to put in some time on your own too. 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.

Dojo Principles

These principles were written by Christophe Thibaut, and first published in Laurent Bossavit's blog[4] in 2005, as a guide for new members of the first dojo, in Paris, France.

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.

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

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

Finding a Master

The master can't be a master of every forms. 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 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 just tries to learn it for the first time. We can learn from others mistakes as well as from ours 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 subject rather than on four different subjects ? 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 time someone begins to master a subject and wants to approach another one. Those threatened by boredom should throw themselves first into a presentation. The goal is to get back to a good motivation level, i.e. an acceptable level of difficulty.

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, to explain it to other who find it difficult. Explain it again as long as they find it difficult.

Dojo Rules

The principles are great, but when you get a bunch of half a dozen coders in a room, you'll quickly find there are at least a dozen opinions on what code to write! These rules are designed to keep the meeting on track, and give everyone the best chance to contribute, teach and learn. It's good to appoint a meeting facilitator, who has a special responsibility to see that these rules are followed.

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.

Dojo Histories

The original Paris Dojo

I'm hoping Laurent Bossavit, Emmanuel Gaillot and/or Christophe Thibaut will kindly write something here about the eight year history of their Dojo in Paris.

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[5], 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[6] (in Europe), or a conference run by the Agile Alliance[7] (in the US). There might be an "XP Day" or a "Software Craftsmanship" conference, or a "Code Retreat" happening nearer where you are.

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 the next section I'll go through some of the practical questions you'll have to sort out when you're organizing your dojo.

Practicalities

First book a date

My strategy when holding a new dojo meeting is to first fix a date and time. Fine one that suits the people you really want to be there, and when you've decided on that, look around for a venue. There's always someone who's got a spare conference room, coffee area or games room[8] that you can borrow.

Meeting Length

If you have less than one hour it becomes difficult to write enough code to learn something new, or have time to discuss it. Two hours is my preference. It gives you time to try out some ideas in code, and for everyone to contribute to the discussions. If you have longer than that, you should

[5]http://meetup.com
[6]http://xp2013.org
[7]http://www.agilealliance.org/
[8]If you can find a trendy startup to sponsor your meeting! (Not that I'm boasting or anything).

break the meeting up into different activities with a short retrospective after each, as well as a longer retrospective at the end.

Some examples:

- The Paris Dojo meetings are around 2 hours in an evening, followed by socializing over a drink or two.
- my local Python user group meetings are 3 hours in an evening, including first eating pizza and discussing Python news. The dojo part is usually about 2 hours.
- Code Retreats are whole day events usually starting at 8:30 am on a Saturday. There are 5 or 6 timeboxed 45 minute coding sessions, with a hot lunch in the middle of the day. (For more information see the chapter on Code Retreat)
- Greg Dziemidowicz runs dojos at his workplace (Nokia) that are 1 hour, using the code retreat format, except with only one 45 minute timeboxed coding session.
- When I go to a company to facilitate dojos with a team, I usually suggest 2 hour meetings.

Meeting frequency

How often should your dojo meet? Well of course the answer is - how often do you have the time and energy? These meetings should be fun to take part in, so you'll want to repeat your dojo. Repetition is essential to learning, too. Real life is likely to intervene and restrict the time you have available for them though. You could aim to meet weekly, biweekly or monthly, or just see these as one off events you hold occasionally.

Some examples:

- The Paris Dojo meets every week[9].
- My local Python user group meets monthly, and about half the meetings take a dojo form.
- In Stockholm we've held Code Retreat events two or three times a year.
- When I'm brought into a company to facilitate dojos with a team I usually suggest biweekly meetings.

Group Size

A coding dojo is no fun all by yourself, so you'll want to get some people to join you. I've never tried a purely virtual coding dojo, and I think an essential element is the rich interaction that happens when people are in the same room. I've had best results with about 5 - 15 people in a room together. Too small and it lacks variety of opinion and style. Too large and discussions become unmanageable. Having said that, I've led successful code retreats with over 40 people

[9]except if the regular meeting night falls on Valentine's Day. They are romantic Frenchmen and -women after all.

present. You just have split into smaller groups while coding, and to be a bit innovative in how you handle retrospectives.

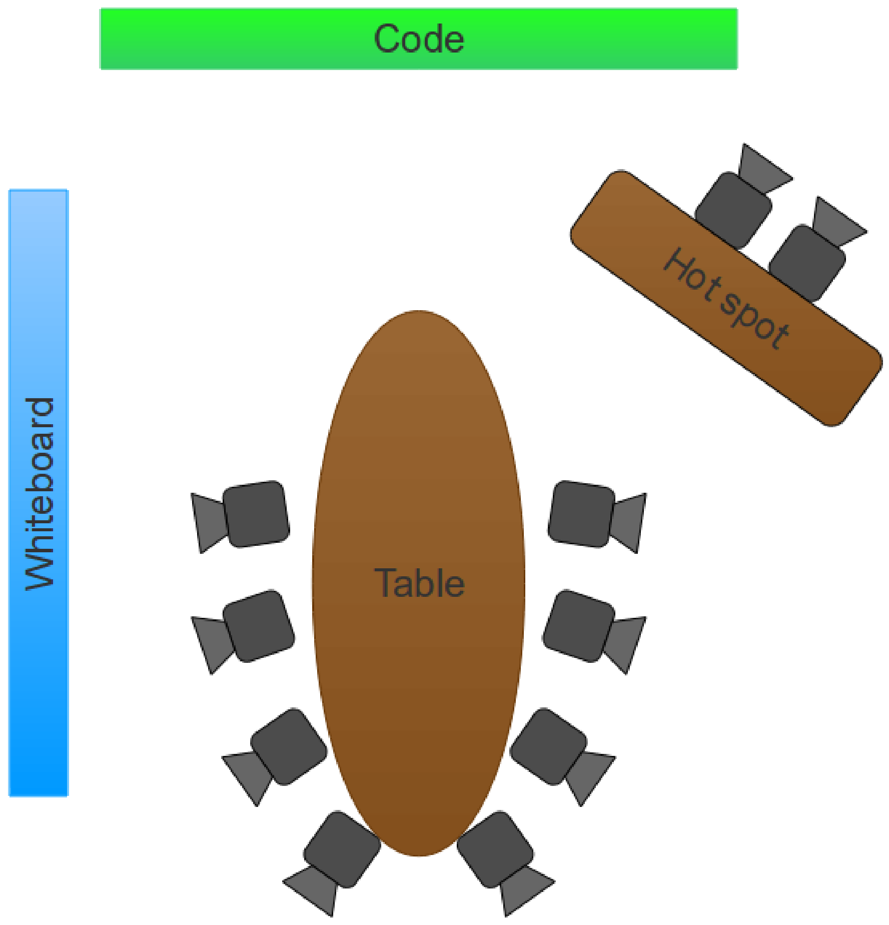
Some examples:

- The Paris Dojo is usually about 4-10 participants.
- My local Python user group is usually about 6-15 participants.
- When I'm brought into a company to facilitate dojos with a team I usually ask them to limit it to 15 people, and we start a second group if there are more teams interested in participating.
- Code Retreats I've facilitated have had 30-45 participants, and we split into two "closing circles" for the retrospective at the end, with one of the ordinary participants stepping in as a second facilitator.
- At conferences I've taken part in dojos with 20, 30, or even over 40 people.

Physical layout of the meeting room

Exactly what you'll need will depend on the activities you've chosen. As a starting point, assume you'll need a projector, at least one computer with a coding environment set up on it, a table to put it on, chairs, and a whiteboard/flipchart to aid discussion. Below is a diagram showing how the room could look[10].

[10]With thanks to Fredrik Wendt for the picture.



Sample room setup

For some activities, (for example a Randori or Prepared Kata), you only need one computer for coding on, set up at the front. You want everyone to be able to see the pair who are coding, the code displayed by the projector, and the whiteboard/flipchart. There should be enough space to move around, write on the whiteboard, and swap the pair who's coding. Often we'll all sit around a large conference table, and when it's time to swap the person coding, everyone gets up and moves one place around the table.

Some people prefer the pair at the front to have a separate table, facing away from the group. It lets them concentrate more on pair programming with each other, since they are less distracted. It can be less scary since it's easier to ignore the "audience". I don't tend to do this though, since you end up with two discussions going on, one between the pair, and another amongst the rest of the group. The group can get sidetracked and stop paying attention to the code being written. I prefer to keep eye contact with everyone.

For other kinds of activities, (for example Randori in pairs, or Cyber-Dojo), you might be working in pairs, and need more tables and computers. Again, a large conference table with pairs sitting all around it works well.

Which Editor or IDE?

Go for an editor and keyboard layout most people are comfortable with if you're going to be switching pairs during coding. Otherwise let each pair decide. If you're practicing a Refactoring Kata you may want people to have their usual refactoring tools available, which often means an IDE. Alternatively you could level the playing field completely by using a tool like Cyber-Dojo, where you edit code in a browser with minimal tooling.

If you're doing a dojo with an international bunch, there are going to be problems with the keyboard layout. Quite frankly, some countries put their curly braces in the most stupid places, and as for quotation marks, underscores and colons, well, let's just say there's no consensus. Some countries (I'm looking at you France) even put the letters in different places!

Dojo Disasters: QWERTY trouble

Once I went to a dojo in Finland, where we were using a French keyboard, using Vi and programming Haskell. Let's just say typing *did* become the bottleneck!

Which Programming Language?

If your focus is on learning something else, like Test Driven Development, you'll want to choose a programming language most people in the group are already comfortable with. It helps you to focus on what you're really interested in learning. If on the other hand you want to learn a particular programming language, you could decide to learn it in the dojo. Take a Kata you already know well in another language, and solve it in the new language. It really helps to have at least one person present who knows the new language well though. Otherwise you spend half the time googling basic syntax and error messages.

Work time or Free time?

If you ask people to give up their free time in order to come to the dojo, you'll likely get a different kind of person coming. They'll often be motivated and enthusiastic about learning, and willing to spend more of their free time in preparation or individual practice. The downside is that the people who really need to learn new coding skills won't ever turn up.

If you can persuade your manager that a coding dojo is a good use of work time, then you might be able to bring your whole coding team into the dojo, even the more reluctant people. In the best case, it will be so fun and rewarding that the reluctant people will become enthusiastic :-)

Another context I've run dojos in is at conferences. It's a different dynamic - still work time - but more enthusiastic people, who usually don't know each other at all. It can be great fun, but as it's something of a one-off event, you don't learn anything very deeply.

Who should you invite?

Well, good programmers who want to become great programmers! Start by asking the kind of people you think you could learn something from to join you. Remember what the Dojo Principles say about mastery? Find people who can broaden your coding horizons. That's not to say that you shouldn't also find some people who want to learn the same things as you. Look at the section Teaching and Learning in the Dojo for ideas of what you could aim to learn together.

Dojo Disasters: Boss Trap

Here I'd like to recount Fredrik's story about having the team's boss in the room during the dojo.

Colleagues or Acquaintances?

If you go to a dojo at a public user group, or a conference, you'll experience coding with some people you have hardly met. You get exposed to their neat tricks with the language and editor and see other ways to code. You might be able to code in a language or environment you're unfamiliar with, or pair program with people who are far more skilled than anyone you normally work with. It can be a rather intimidating experience though. And you might have to give up your free time to do it.

If you go to a dojo with people you already work with then you'll probably still learn some new neat tricks with the language and editor, and you might still get to try out a coding language and environment you're unfamiliar with. The real benefit of it though is that you can bring up issues and discussions that you never normally have time or energy for, and try to get a team consensus around them. Should we be writing the tests first? What are our coding standards? How do we want to use mocks?

You're working on toy code that no-one has a big personal stake in. You know it may be preserved in a repository branch somewhere but you're not still going to be maintaining it in 5 years time. It's a safe environment to bring up issues, discuss them, and hopefully change the way you work in your production code afterwards.

Meeting Activities

The Coding Dojo is all about creating a safe, fun environment for learning, while coding. There are many ways to achieve that, and this chapter is about some popular activities that fit into the basic meeting format and aims. If you're just starting a new Dojo, I'd suggest trying a Randori format first. It involves the whole group working together, and helps you get to know one another. The other formats are all valuable in different ways, and provide variety for your dojo.

Randori

Work on a Kata with everyone together, taking in turns to type. No-one need have done the kata before.

- Create a new empty coding project. Check the coding environment is set up correctly by writing and executing an empty failing test.
- Discuss the kata and decide a general approach. Perhaps make a list of test cases on the whiteboard.
- Decide a mechanism for regular switching of the driver and copilot, see Pair Switching Strategies
- Two people step up to the keyboard and begin writing tests and code
- Use Test Driven Development
- Remember to switch the pair at the keyboard according to the mechanism chosen
- Everyone present is expected to follow what is going on, and make helpful suggestions.
- The pair at the keyboard should explain what they are doing so everyone can follow.
- The audience should give advice/suggest refactorings primarily when all the tests pass. At other times the pair at the keyboard may ask not to be interrupted.

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, 5 minutes for Python.

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 Disasters: 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 small 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, if they get the keyboard.

NTests

The pair at the keyboard write and implement N tests, where N is usually between 1 and 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.

ShoutLouder

- The pair at the keyboard are unsure what direction to take.
- Someone in the audience suggests a direction.
- The pair ask that person to take over driving, the driver becomes co-pilot, the co-pilot sits down in the audience

As a mechanism for pair switching, this one is a bit dangerous. It works better if the pair at the front can choose not to hand over the keyboard, just take the advice. Then the problem is that some pairs will never hand over the keyboard to anyone else.

Dojo Disasters: Shouting Competition

"I have seen this grind a dojo to a halt as people haul the design back and forth in competing directions" - Matt Wynne - I think you have a story about this. Would you like to tell it here?

Micropairing[11]

The idea is that you switch pairs after a fixed time, as with Timebox. Within each timebox, you pass the keyboard each time one of the following things happen:

- Driver writes a failing test.
- Driver makes a failing test pass.
- Driver refactors something.
- Driver writes a passing test. (This happens sometimes, even with TDD)

This approach can be especially helpful if people are new to pair programming. It makes sure that both people get involved with refactoring.

[11]This approach was first proposed by Peter Provost

Prepared Kata Performance

You can learn a lot by watching an expert at work. You can learn a lot when teaching. In a prepared Kata demonstration, someone has practiced the kata many times, and is showing the group their best solution. Not just the finished code, but the entire process from empty editor via Test Driven Development to a full working solution. As they code, they should explain their reasoning and choices, so everyone can follow what is happening and why the code turns out like it does.

The person at the keyboard is setting themselves up in a particularly vulnerable position, and it takes quite a bit of courage. It's not easy to code in front of an audience and talk at the same time. You will usually choose a pairing partner from the audience to support you. This person has a particular responsibility to point out omissions, typos etc, but actually everyone in the room should try to be supportive and kind. Comments and suggestions for improvements can be experienced as distracting though, so you're free to ask people to save them for the retrospective.

If you are new to kata performance, it can be less stressful to prepare together with a pairing partner. You can learn a lot from your pair during practice sessions. When it comes to the performance, you'll be able to demonstrate good pair programming in action as well as TDD.

If you're in the audience watching a Prepared Kata performance, your first priority is to make sure you understand what's happening. The ideas is that you follow in enough detail that you'll be able to go home and reproduce the whole Kata for yourself afterwards. The pair at the front should always be willing to stop and explain their thinking. Your other job is to try to think of better ways to do the Kata. Did the pair produce a good design in the end? Are they taking small steps, especially when refactoring? Would a different order of tests lead to better design insights?

You should set a time limit for the performance, to be sure there is enough time for the retrospective. At the retrospective, everyone should give feedback and constructive criticism. Both the performers and the audience should have learnt something. Everyone should be able to go away and do the Kata better the next time.

Randori In Pairs

Split into pairs (or trios) and work on a Kata. No-one need have done the kata before. This is a good alternative if the group is getting a bit large for an ordinary Randori. With more than about 10 people, the discussions can get out of hand, and each individual doesn't get much time at the keyboard.

This Randori variant works well for learning from your pairing partner, but you need some way to share what you've learnt with the group. There are a couple of variants:

- after coding, plug each computer in turn into the projector, and have each pair explain how they wrote it.
- switch pairs during coding. After 10 minutes, ring a bell and have one half of each pair get up and go to a different computer.

Usually you'll have all the pairs working on the same kata in the same programming language. You could also try it with each pair using a different programming language for the same Kata. This only works if a critical mass of people know each language in use.

Dojo Disasters: It's all Greek to me

One time I was at a conference, and we decided to do a Randori in Pairs, switching pairs every 10 minutes. Each pair got to set up their own environment, and choose a programming language. Of course at the time Ruby was the next hot thing, and although almost everyone there was a Java programmer, half the pairs chose Ruby, or even something more sexy, like Clojure or Erlang. As we circulated around, it became clear that hardly anyone knew what they were doing in these languages, and there was much hacking, googling and tearing of hair. It was actually pretty boring, unless you happened to be pairing with one of the few polyglot gurus!

Following that conference, and similar experiences, Jon Jagger actually changed the way you set up a new Cyber-Dojo session so that all participants have to use the same programming language.

Using Production code in the dojo

Normally all the code you work on in the dojo is toy code that you will throw away afterwards. Here I'll talk about situations where you might bring production code into the dojo.

When you know enough TDD that Katas seem easy

If you can do TDD well on code Katas then hopefully you'll be able to start doing it in your production code in everyday work. To smooth the transition, some teams like to hold dojos where they work on production code together. It can be so informal as that the team just pull the next story to work on from the task board, and take it into the Dojo. They follow the usual meeting format, including explanations, planning and retrospective, and work on the production code all together in Randori format. It can be a chance to get the whole team talking about real issues in the real codebase. You just have to be very careful to keep the atmosphere safe and constructive, keep doing TDD, and not disappear down any rabbit holes. (By that I mean continue to make progress in small steps, with tests, and not try to refactor the whole codebase in one go).

Identify a suitable Kata from production code

Sometimes when you're working on production code, you find TDD working really well, and that the piece of code you're building feels small and self contained like a Kata. You might want to package it as such, and try it out at your next dojo. Others can get the benefit of practicing on a Kata where TDD works well, but yet is drawn from a "real" situation. Remember that for an exercise to work well as a Kata, it needs to be quite small and self contained, so that you can code it from scratch in 1-2 hours. There are a few Katas in the catalogue that arose this way.

This Code is impossible to TDD!

Alternatively, you might be finding TDD really hard with a particular piece of legacy code. Get your most experienced developer to do some preparation, and practice TDD:ing a tricky piece of code. They can then show this as a prepared Kata to the rest of the team. The idea is to inspire everyone that TDD is possible after all.

Dojo Disasters: Norwegian Blue(s)[a]

Here I'd like to include Johannes' story about the team who wanted him to show them doing TDD in their production code, then asked him to stop, because it was too depressing.

[a] Lovely plumage! (For all you Python programmers out there)

Skill levels

Here I will write something about how to change the dojo meeting depending on the coding skill levels of the participants.

Code Retreat

Here I will explain what Code Retreat is and how you can run one yourself. Also how it differs from coding dojo.

Tools for the Dojo

Here I will say something about Cyber-Dojo, CoderDojo client, and possibly Sessions.

Cyber-Dojo

Handling Critical Voices

Sometimes in the Dojo, you'll find people start to question what's going on. They start to be critical, even hostile to the idea of practicing with Katas, ridicule the idea of Test Driven Development, and dismiss what you're doing as "messing about with toy code". People can get angry and make personal attacks. This can destroy the feeling of safety necessary for learning, at the same time as some criticisms are valid and need to be addressed.

The most important thing is to stay calm yourself, and keep treating everyone with respect, even if others are being rude. Try to be lighthearted, perhaps make a joke to try to diffuse the tension. You could even try apologising. No really, even if you personally haven't done anything wrong, it can change the dynamic. (We British are very good at apologising, if you step on a British person's foot I can almost guarantee they'll say "sorry!").

You could try reminding people about the Dojo Rules or the Dojo Principles. You could ask people to save their comments until the Retrospective, (by which time they'll have calmed down). You could write it up on the whiteboard as a "Parked topic" that you don't want to address right now. In the worst case, you could ask someone to leave the room, or break off the meeting altogether. Thankfully I've never yet found I needed to go that far.

Herding cats

It was 2006, not long after Laurent Bossavit had visited my (then) workplace to teach us about the dojo, and I was really keen to try facilitating one. I discovered there was a newly-formed Gothenburg Ruby User Group, (a language I didn't actually know), and enthusiastically suggested to their leader (who I had never actually met) that we could do a dojo at the next meeting. Surprisingly, he agreed, and one rainy Wednesday evening after work, I met him at a small office above a shop somewhere on a Gothenburg backstreet. I set about reorganizing the furniture, while he set up his mac at the front, (a platform I'd never coded on). Shortly afterwards, roughly 20 ruby coders, (all of whom were men, few of whom I'd met before), turned up and started tucking into the free pizza on offer.

I talked a bit about the dojo idea, introduced the Kata, and then we started coding. I very rapidly learnt that the mac has a rather useful "apple" or "bun" key, and all sorts of exciting things happen if you try to copy and paste text using the "Ctrl" key. Thankfully in the end someone was also kind enough to tell me when the Python syntax I wrote didn't actually work in Ruby. I heaved a sigh of relief when the timer pinged for the end of my timebox driving, and I handed over the keyboard to the next person. For a while everything seemed to be going rather well, lots of people were talking, and tests and code were being written.

Then some people started asking challenging questions. Why are we writing tests? It would be much quicker/better if we did it my way. This design sucks. 5 minutes is too short at the keyboard. I don't want to "fake it", why would I write something I'm just going to delete in a moment? I can write this code much better than (person who is currently driving), give me the keyboard! We should be using metaprogramming! (Instead of that boring code there that just made the tests pass). Have you seen my latest open source project? I've done this really cool thing…

I had never had to moderate this kind of discussion before, and I was lacking a lot of answers, shall we say. I was very glad when various of the more experienced Rubyists joined in, defending TDD and Refactoring and helping me keep the meeting on course. Thinking back, it was a kind of jostling for position and geek status, amongst a group of young programmers who didn't know each other very well. I didn't have the Dojo Rules in place (I don't think they were invented then!) and I'd stepped well outside my comfort zone in a few too many ways. At the time it felt very much like I'd spent the evening herding cats.

I survived, and lived to lead another dojo, but I wouldn't recommend it!

Common Objections

Here are some of the most common objections that come up, and some thoughts about how to handle them.

Code Katas are just toy code, they don't teach you anything useful.

Katas absolutely **are** toy code. It's not going into production, it's probably implemented better somewhere on the internet, and you'll likely throw your code away straight after the Dojo. That's why it's fun to work on!

Doing a Code Kata can absolutely teach you something. Learning new skills like TDD is hard. It's even harder if you're working on a real world coding problem you haven't faced before. Code Katas let you focus on everything *other* than delivering working code to customers. You can relax, try stuff out, make mistakes, throw it out and start over. If you experience the rhythm and flow of TDD in the ideal circumstances of a Code Kata, you might recognize that feeling when you get it in production code. You might even seek it out.

Why would I want to "Fake it" as a step in TDD? It doesn't make sense to write code I know I'm going to delete straight away.

If you can see what to write for the real solution, just write it! "Fake it" is a strategy in TDD to help you when you can't see the full, general thing to write straight away. It helps you via "Triangulation" to reach that general solution. In a code Kata it's often really easy to jump straight to the general solution, but the idea is, that we're practicing on easy code so we'll know what to do when things get tricky. Perhaps we could try that today?

Also re-read what Kent Beck writes about "Fake it ‘til you make it" in his book "Test Driven Development by Example". (It's a Green Bar Pattern)

I can do TDD in Code Katas but it doesn't help me do it in my production code.

If you can do TDD in a Code Kata you're doing better than 99% of the programmers out there! Seriously! The step to doing it in your production code is probably about learning better design techniques. Some designs are inherently harder to test, and unfortunately there's a lot of that about. It could be that you've been practicing TDD on Katas where it's quite easy to find the tests to write and build up the design gradually, and your production code isn't like that. Have a look around the chapter in this book about Using Code Katas To Learn TDD. Try to find some Katas that help you practice around the problems you're seeing when trying to test drive your production code.

If you like, you could try bringing your production code into the dojo. Maybe someone there can help you see what to do differently. See the chapter on Using Production Code in the Dojo

Having said that, there are some situations when TDD is more difficult to do, or might not be the best choice. For example when designing a Graphical User Interface, or multithreaded code. If you think you're in that kind of a situation, relax. Choose another testing approach, then try TDD again when the situation seems more appropriate.

You're taking such small steps! No-one codes like that in the real world, it would be too slow.

Yeah, I guess it might be a bit boring to watch someone coding so slowly. I think that's a big problem with good code, actually. It looks boring! The point is that sometimes you're working on really tricky code and you **do** need to take small steps. If you practice taking small steps on easy code, then you'll have a better chance of being able to do it when things get difficult. Actually, you can make very good progress even with small steps, and it's usually a less stressful way to work. Remember the fable of the tortoise and hare?

I don't believe TDD is a useful technique. I use better ways to produce good code.

If you have better ways to produce good code then I expect we'd all like to learn about them! Perhaps you could give us a prepared Kata demonstration sometime?

Actually, I'm a little bit skeptical when you say that you don't think TDD is a useful technique. I've heard some very good programmers use it, quite a lot of the time. Perhaps we should learn more about it before we decide. Can we spend the next few dojo meetings learning about TDD then reassess if we'd be better off moving on to other techniques?

Remember the Dojo Principle about mastery? If you're a master of some other technique, you still may still find new knowledge and mastery together with others in the dojo. Please stay and we'll all learn something!

Section 2: Teaching & Learning In the Dojo

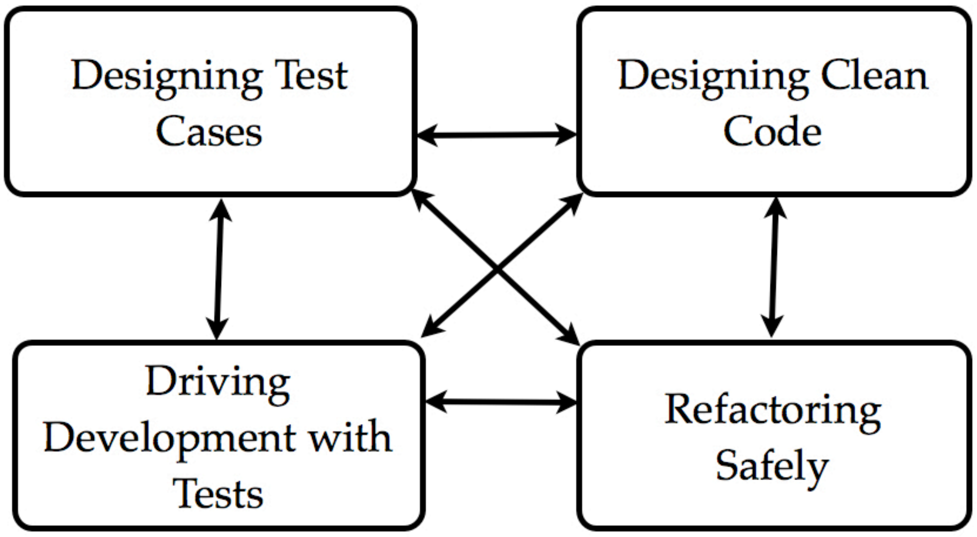
This section is all about the kinds of skills you can practice in the Dojo. One of the key skills you're trying to improve at in the dojo is Test Driven Development. Indeed one of the Dojo Principles says *"code without tests simply doesn't exist."* So much of this section is about TDD - there are lots of Katas you can practice on. Later in this section I'll look at Katas that help you to learn Functional Programming idioms, and other skills.

Teaching & Learning Test Driven Development

The first part of this chapter gives an overview of what katas you could use to learn different aspects of TDD. The rest explains a little about how I talk about TDD in the dojo, and what I encourage people to focus on. I've chosen to supplement the classic texts with a couple of articles of my own. You may already feel quite happy about the idea of teaching TDD to beginners. In that case you might want to skip over the rest of this chapter.

Using Code Katas to learn TDD

Test Driven Development is a multifaceted skill, that it takes study and practice to master. I think you can break it down into four closely related, and mutually supportive sub-skills:



TDD sub-skills

When you're learning TDD it can help to "home in" on one or two of these sub-skills, and choose a Kata that challenges you more in that area. Here I've listed Katas that are good for each sub-skill, roughly in order of difficulty. Over a series of dojos, you could aim to do Katas which stretch you in each area in turn.

In addition, there are other styles of TDD that you can learn once you feel you're on top of the classic TDD skills. Perhaps the most well known is the London School of TDD. You could also look at [Approval or Text-Based Testing], which can be particularly useful in the context of legacy code.

Driving Development with Tests

This skill is about gradually building up a piece of code to solve a problem that's too big to solve in one go. You start with a Guiding Test as a goal, but that test should be too difficult to make pass in one step. You need to break down the problem into small pieces, identify test cases, and choose what order to implement them in. I think Kent Beck explains this process well in his book "Test Driven Development by Example".

When you're practicing these katas, see if you can begin with tests that only require a very simple implementation, but that should continue to pass even as you write more tests and build up an implementation for the whole problem.

- StringCalculator
- FizzBuzz
- Yahtzee
- Bowling

Refactoring Safely

This skill is about making a sequence of tiny, safe changes, that add up to a larger design improvement. The classic text on this is by Martin Fowler, "Refactoring: Improving the design of existing code".

All these katas have starting code that is less than clean, and your task is to improve the design without breaking the functionality. Some of these katas provide test cases, for others you also have to design the test cases you'll need to lean on while refactoring.

- Tennis
- Yahtzee
- GildedRose
- Four Katas on a SOLID Theme

Designing Test Cases

This skill is about designing test cases that cover the functionality, while at the same time being readable, robust and fast to execute. Beginners often enthusiastically write lots of tests, and later discover they are expensive to maintain, and rarely catch real problems. I've written about four Principles for Agile Automated Test Design in the next chapter. Alternatively, look at the "Three Pillars of Good Tests" from "The Art of Unit Testing" by Roy Osherove.

When you practice these katas, review your test cases carefully and discuss how well they follow the principles and/or pillars.

- GildedRose
- Minesweeper
- Reversi
- Medicine Clash

Designing Clean Code

This skill is about code that is readable, and makes good use of language idiom and style. For Java in particular, Robert Martin's book "Clean Code" is a pretty good guide. His book "Agile Software Development: Principles, Patterns and Practices" explains the SOLID design principles. Otherwise, a text like "Working Effectively with Legacy Code" by Michael Feathers can give you some good ideas.

- Four Katas on a SOLID Theme
- Medicine Clash

TDD in terms of States and Moves

People often use Robert Martin's "3 rules"[12] to explain TDD:

1. You are not allowed to write any production code unless it is to make a failing unit test pass.
2. You are not allowed to write any more of a unit test than is sufficient to fail; and compilation failures are failures.
3. You are not allowed to write any more production code than is sufficient to pass the one failing unit test.

Although this is undoubtedly a correct description, I find it a little terse and fierce to use with complete beginners. I want people to feel safe in the dojo, and have instructions that help tell them positively what they should be doing.

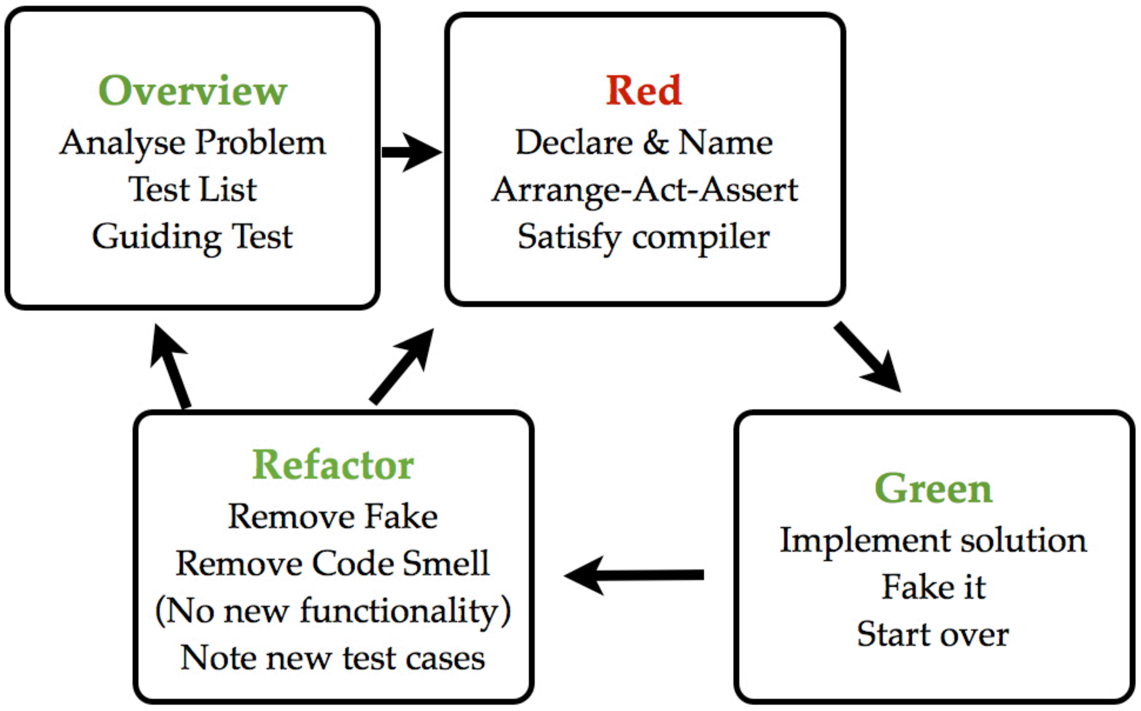
Dojo Disaster: Facepalm

I was facilitating a dojo for a small group, where a couple of the people hadn't even heard of TDD before. We were doing a Randori, with Ping-Pong switching. After we'd been coding only a little while, someone slightly more experienced wrote a failing test, then handed the keyboard to one of these beginners. The beginner looks puzzled and starts reading the code. "What should I do?" he asks. "Do the simplest thing possible to make the test pass!" comes the suggestion. So he goes to the test code, finds the failing test, and changes the "expected" value in the assertEquals to match the "actual" value. "Look! it passes!". (Collective facepalm).

So that's what gave me the idea for TDD in terms of states and moves. When you're in the Red state, generally the only valid moves involve changing the *production* code. I've found explaining this beforehand reduces embarrassment at the keyboard.

My preferred way to explain TDD to beginners uses the picture below. Most people will be familiar with the Red-Green-Refactor cycle, but I've also added an "Overview" state.

[12]http://butunclebob.com/ArticleS.UncleBob.TheThreeRulesOfTdd



TDD in terms of States and Moves

In the Dojo, if there are TDD beginners present, I'll normally sketch this picture up on a whiteboard before we start coding. As I write each state, I'll explain a bit about what I mean. Let me take you through my TDD explanation.

Overview State

Before you start coding, I think it pays to step back a little and try to analyse the problem we're trying to solve. Not long - we don't want analysis paralysis or Big Design Up Front. In the Dojo, when you get to the coding part, I'd suggest actually spending up to 15 minutes talking about the chosen Kata before writing much (any!) code. You need to help everyone understand the problem you're planning to solve together. Start by sketching and noting test cases on the whiteboard.

In the **Overview** state you:

- Make a basic analysis of the problem.
- Review any existing code in area and identify where the new functionality will be called from.

- Go back to the Customer/Product Owner and ask questions.
- Write a Guiding Test.
- Make a list of other (smaller) test cases, and think about what order to implement them in.

Move on to the next state - **Red** - when:

- You understand your goal for the coding session & have chosen your next test

Clarifications

The Customer or Product Owner is the person you as a developer talk to in order to clarify requirements. In the Dojo, usually the facilitator takes this role.

A "Guiding Test" is a test that will pass when you're done with your coding session, but is too big to make pass straight away. It gives you a goal for your session, and is usually a non-trivial example use for your User Story or Kata. Sometimes people call it a "Coaching Test" since it should coach you into the right frame of mind to tackle the problem. Others call it an "Acceptance Test" since it can reflect acceptance criteria for a User Story. Still others prefer to call it a "Story Test" since it is a test for a whole User Story. In Behaviour Driven Development it could be a Scenario expressed as a unit test.

Red state

You're trying to set up a small, achievable goal. Choose a test from your list, that will help you towards your session goal (Guiding Test), or will force you to address a weakness in your current production code. If you've just made a fake to get the last test to pass, this next test should force you to remove it (via what Kent Beck calls "triangulation"[13]).

In the **Red** state you:

- Declare and name a test
- Arrange - Act - Assert (start by writing the Assert)
- Add just enough production code to satisfy compiler (if you have one)

Move on to the next state - **Green** - when:

- Your test describes the missing functionality, executes, and fails.

[13]in his book "Test-Driven Development by Example"

Clarifications

When writing the test name, you might find it helpful to use this template:

- method_name_With_arguments_Returns_return_value()

for example:

- add_With_Empty_String_Returns_Zero()

(Which could be the first test in the StringCalculator Kata)

Remembering the 3 'A's "Arrange - Act - Assert" helps you to structure your actual test code. Bill Wake invented the idea, and wrote an article that I think explains it well: "3A - Arrange, Act, Assert"[14]

Green state

You're trying to get the new test to pass as soon as possible, even if that means taking shortcuts with the design. Copy and paste code if you must. Make a long method even longer. Fake it. On the other hand, if you can see how to do it right, just do it. Just don't take too long over it. You should try to never be more than a few minutes away from having all the tests passing. Even if that means deleting all the code and the test you just wrote.

- Change the production code - implement just enough to make the test pass
- Fake it (if you're unsure)
- When all else fails, remove the failing test, get back to green, and write an easier test.

Move on to the next state - **Refactor** - when:

- Your tests all execute and **pass**

Refactor state

Now you have the system working, make it right. Remove duplication and other code smells. Take small steps so you are never more than a few keystrokes from having all the tests passing. Don't forget to also refactor the test code to the same standard as the production code.

- Change production code: remove fakes, code smells

[14]http://xp123.com/articles/3a-arrange-act-assert/

- Change test code: improve readability
- Don't add functionality not required by tests.
- You might think of new test cases while you're refactoring - note them on your list.
- Run the tests after each refactoring. They should continue to pass.

Move on to the next state - **Red** or **Overview** - when:

- The code is clean and your tests all execute and pass.
- Move to the "Red" state if you already know the next test to write. Otherwise move to the "Overview" state.

Clarifications

While you're refactoring, you might make a mistake. In this case hopefully the tests will alert you by failing. In this case the safest thing to do is just to undo all your recent changes until you get back to passing tests. If you can easily see the mistake you've made though, do go ahead and just fix the problem. If you can't quickly see the problem, and start hacking about, you're on a slippery slope. Hopefully your pairing partner won't let you continue sliding down it, but will force you to back out your failed refactoring to get quickly back to the solid ground of passing tests.

While you're refactoring, you will probably spot weaknesses in the code - functionality that is still missing, or maybe hard coded fakes we forgot about. Note these things down as tests on the list. If you fix them straight off the danger is you won't have test coverage for the increased functionality.

When to stop

At some point hopefully you'll get your guiding test to pass, and you'll be happy with the design of your code. Often in the dojo, the time runs out for coding before this happens. Try to resist the temptation to skip the retrospective and just keep coding. It's important to reflect on what's happened and discuss what you'd do differently if you did this exercise again. At your next dojo, you can always do the same Kata again. You'll start from scratch of course, but you should get further since you've learnt more about the exercise.

Principles for Agile Automated Test Design

Once you've worked on a system with extensive automated tests, I don't think you'll want to go back to working without them. You get this incredible sense of freedom to change the code, refactor, and keep making frequent releases to end users. However you arrive at those tests (TDD, test-first, test-last, LondonSchool, whatever), I think the test code needs to follow certain principles if the tests are going to be useful in the longer term.

I think the same properties are needed for good agile functional, integration and system tests as for good unit tests, but it's much harder. Your mistakes are amplified as the scope of the test increases. It's good to learn to write well designed tests at the unit level before you move on to larger-scale tests. The dojo is a good place to do that!

I'd like to outline four principles of agile test automation that I've derived from my experience.

Coverage

If you have a test for a feature, and there is a bug in that feature, the test should fail. Note I'm talking about coverage of functionality, not code coverage, although these concepts are related. If your code coverage is poor, your functionality coverage is likely also to be poor.

If your tests have poor coverage, they will continue to pass even when your system is broken and functionality unusable. This can happen if you have missed out needed test cases, or when your test cases don't check properly what the system actually did. The consequences of poor coverage is that you can't refactor with confidence, and need to do additional (manual) testing before release.

The aim for automated regression tests is good Coverage: If you break something important and no tests fail, your test coverage is not good enough. All the other principles are in tension with this one - improving Coverage will often impair the others.

Readability

When you look at the test case, you can read it through and understand what the test is for. You can see what the expected behaviour is, and what aspects of it are covered by the test. When the test fails, you can quickly see what is broken.

If your test case is not readable, it will not be useful. When it fails you will have to dig though other sources outside of the test case to find out what is wrong. Quite likely you will not understand what is wrong and you will rewrite the test to check for something else, or simply delete it.

As you improve Coverage, you will likely add more and more test cases. Each one may be fairly readable on its own, but taken all together it can become hard to navigate and get an overview.

Robustness

When a test fails, it means the functionality it tests is broken, or at least is behaving significantly differently from before. You need to take action to correct the system or update the test to account for the new behaviour. Fragile tests are the opposite of Robust: they fail often for no good reason.

Aspects of Robustness you often run into are tests that are not isolated from one another, duplication between test cases, and flickering tests. If you run a test by itself and it passes, but fails in a suite together with other tests, then you have an isolation problem. If you have one broken feature and it causes a large number of test failures, you have duplication between test cases. If you have a test that fails in one test run, then passes in the next when nothing changed, you have a flickering test.

If your tests often fail for no good reason, you will start to ignore them. Quite likely there will be real failures hiding amongst all the false ones, and the danger is you will not see them.

As you improve Coverage you'll want to add more checks for details of your system. This will give your tests more and more reasons to fail.

Speed

As an agile developer you run the tests frequently. Both (a) every time you build the system, and (b) before you check in changes, and (c) by the CI server after checkin. I recommend time limits of 2 minutes for (a), 10 minutes for (b), and 45 minutes for (c). This fast feedback gives you the best chance of actually being willing to run the tests, and to find defects when they're cheapest to fix.

If your test suite is slow, it will not be used. When you're feeling stressed, you'll skip running them, and problem code will enter the system. In the worst case the test suite will never become green. You'll fix the one or two problems in a given run and kick off a new test run, but in the meantime someone else has checked in other changes, and the new run is not green either. You're developing all the while the tests are running, and they never quite catch up. This can become pretty demoralizing.

As you improve Coverage, you add more test cases, and this will naturally increase the execution time for the whole test suite.

How are these principles useful?

I find it useful to remember these principles when designing test cases. I may need to make tradeoffs between them, and it helps just to step back and assess how I'm doing on each principle from time to time as I develop.

I also find these principles useful when I'm trying to diagnose why a test suite is not being useful to a development team, especially if things have got so bad they have stopped maintaining it.

I can often identify which principle(s) the team has missed, and advise how to refactor the test suite to compensate.

For example, if the problem is lack of Speed you have some options and tradeoffs to make:

- Invest in hardware and run tests in parallel (costs $)
- Use a profiler to optimize the tests for speed the same as you would production code (may affect Readability)
- Push down tests to a lower level of granularity where they can execute faster. (may reduce Coverage and/or increase Readability)
- Identify key test cases for essential functionality and remove the other test cases. (sacrifice Coverage to get Speed)

Explaining these principles can promote useful discussions with people new to agile, particularly testers. The test suite is a resource used by many agile teamembers - developers, analysts, managers etc, in its role as "Living Documentation" for the system, (See Gojko Adzic's book "Specification By Example"). This emphasizes the need for both Readability and Coverage. Automated tests in agile are quite different from in a traditional process, since they are run continually throughout the process, not just at the end. I've found many traditional automation approaches don't lead to enough Speed and Robustness to support agile development.

Other Principles

In his book "The Art of Unit Testing", Roy Osherove talks about "Three Pillars of Good Tests", which I'd summarize like this:

- **Trustworthiness**: test results are accurate, developers will accept them with confidence.
- **Maintainability**: tests are easy and quick to update when code changes.
- **Readability**: you can easily and correctly interpret test failures.

Obviously we agree on the importance of Readability. I think his other two pillars are a mixture of my other three principles. If you prefer to think about the three pillars instead of four principles, then don't let me stop you! I think our goal is the same.

The London School of TDD

The London School encourages you to design test cases that make extensive use of Mock Objects to isolate the class under test. You end up checking interactions between objects, rather than only focussing on checking object state.

(Here I hope to expand on this description of the London School)

Many Code Katas only need you to write one class or one public method, and checking for state is perfectly sufficient to fully test them. There are a few katas that are actually easier to do in the London style though.

- Four Katas on a SOLID Theme
- MontyHall

Approval or Text-Based Testing

Here I will write something about what this is and what katas you might use to learn it.

- GildedRose
- Trivia

Functional Programming

Code Katas can be solved in many ways, and some lend themselves to a functional paradigm. If you'd like to learn a functional programming language, or just learn to use a more functional style in a multi-paradigm language like Python, there are some Katas that are better suited. Look for opportunities for recursion, and operations like map, filter, sum. Also think carefully about datastructures and how to handle state.

- Bank OCR
- Game of Life
- Reversi

Behaviour Driven Development, Specification by Example, ATDD

Here I will write about some Katas you can use to learn more about the tests you write as part of these approaches.

Section 3: Kata Catalogue

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.

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.

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. There are also standard questions you can ask for almost any kata:

- Are we happy with the design of the code?
- Did we use Test Driven Development well?
- Did we learn anything new?
- Did anything unexpected happen?
- What do we still need to practice more?
- What should we do differently in the next dojo?

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: Args

Most of us have had to parse command-line arguments from time to time. If we don't have a convenient utility, then we simply walk the array of strings that is passed into the main function. There are several good utilities available from various sources, but they probably don't do exactly what we want. So let's write another one!

The arguments passed to the program consist of flags and values. Flags should be one character, preceded by a minus sign. Each flag should have zero, or one value associated with it.

You should write a parser for this kind of arguments. This parser takes a **schema** detailing what arguments the program expects. The schema specifies the number and types of flags and values the program expects.

Once the schema has been specified, the program should pass the actual argument list to the args parser. It will verify that the arguments match the schema. The program can then ask the args parser for each of the values, using the names of the flags. The values are returned with the correct types, as specified in the schema.

For example if the program is to be called with these arguments:

```
-l -p 8080 -d /usr/logs
```

this indicates a schema with 3 flags: l, p, d. The "l" (logging) flag has no values associated with it, it is a boolean flag, True if present, False if not. the "p" (port) flag has an integer value, and the "d" (directory) flag has a string value.

If a flag mentioned in the schema is missing in the arguments, a suitable default value should be returned. For example "False" for a boolean, 0 for a number, and "" for a string.

If the arguments given do not match the schema, it is important that a good error message is given, explaining exactly what is wrong.

If you are feeling ambitious, extend your code to support lists eg

```
-g this,is,a,list -d 1,2,-3,5
```

So the "g" flag indicates a list of strings, ["this", "is", "a", "list"] and the "d" flag indicates a list of integers, [1, 2, -3, 5].

Make sure your code is extensible, in that it is straightforward and obvious how to add new types of values.

Notes:

- What the schema should look like and how to specify it is deliberately left vague in the Kata description. An important part of the Kata is to design a concise yet readable format for it.
- Make sure you have a test with a negative integer (confusing - sign)

- The order of the arguments need not match the order given in the schema.
- Don't forget to check suitable default values are correctly assigned if flags given in the schema are missing in the args given.

Additional discussion points for the Retrospective

- Compare your api with the api of the standard command line argument parsing package you'd normally use in your language.
- Is your error handling code hiding the logic of the main flow? (You do have error handling code, right?)

Ideas for after the dojo

In Robert C. Martin's book "Clean Code" there is a full worked solution written in Java. He mentions in a footnote on page 200 that he has also solved it in Ruby. The code is available on his github page here for Java[15], and here for Ruby[16]. Review the code. How can the Ruby version be so much smaller?

Try the Kata again yourself in a different programming language. Did it turn out smaller than the first time?

Contexts to use this Kata

This Kata is about writing a simple parser, and designing a library API. Othere Katas that involve parsing strings are for example StringCalculator and BankOCR.

If you're reading the book "Clean Code" by Robert C. Martin, I recommend you do this kata for yourself before you read the chapter on it in the book. I think this Kata is interesting to do in different kinds of programming languages and compare your choices of api and schema definition. Also compare the readability/size of the code you end up with.

[15]http://github.com/unclebob/javaargs/tree/master
[16]http://github.com/unclebob/rubyargs/tree/master

Kata: Bank OCR

User Story 1

You work for a bank, which has recently purchased a spiffy machine to assist in reading letters and faxes sent in by branch offices. The machine scans the paper documents, and produces a file with a number of entries which each look like this:

```
    _  _     _  _  _  _  _
  | _| _||_||_ |_   ||_||_|
  ||_  _|  | _||_|  ||_| _|
```

Each entry is 4 lines long, and each line has 27 characters. The first 3 lines of each entry contain an account number written using pipes and underscores, and the fourth line is blank. Each account number should have 9 digits, all of which should be in the range 0-9. A normal file contains around 500 entries.

Your first task is to write a program that can take this file and parse it into actual account numbers.

User Story 2

Having done that, you quickly realize that the spiffy machine is not in fact infallible. Sometimes it goes wrong in its scanning. The next step therefore is to validate that the numbers you read are in fact valid account numbers. A valid account number has a valid checksum. This can be calculated as follows:

```
account number:  3  4  5  8  8  2  8  6  5
position names:  d9 d8 d7 d6 d5 d4 d3 d2 d1
```

checksum calculation:

```
(d1+2*d2+3*d3 +..+9*d9) mod 11 = 0
```

So now you should also write some code that calculates the checksum for a given number, and identifies if it is a valid account number.

User Story 3

Your boss is keen to see your results. He asks you to write out a file of your findings, one for each input file, in this format:

```
457508000
664371495 ERR
86110??36 ILL
```

ie the file has one account number per row. If some characters are illegible, they are replaced by a ?. In the case of a wrong checksum, or illegible number, this is noted in a second column indicating status.

User Story 4

It turns out that often when a number comes back as ERR or ILL it is because the scanner has failed to pick up on one pipe or underscore for one of the figures. For example

```
    _  _  _  _  _  _     _ 
|_||_|| || ||_   |  |  ||_ 
  | _||_||_||_|  |  |  | _|
```

The 9 could be an 8 if the scanner had missed one |. Or the 0 could be an 8. Or the 1 could be a 7. The 5 could be a 9 or 6. So your next task is to look at numbers that have come back as ERR or ILL, and try to guess what they should be, by adding or removing just one pipe or underscore. If there is only one possible number with a valid checksum, then use that. If there are several options, the status should be AMB. If you still can't work out what it should be, the status should be reported ILL.

Additional discussion points for the Retrospective

- How readable are your test cases? Can you look at them and easily see which digits are being parsed?
- What would happen if the input changed format to 12 digits instead of 8? How well would your code cope?

Ideas for after the dojo

This Kata is too big for just one meeting, you'll probably need several to get to all four parts. If you're interested in experimenting with a functional paradigm, try this Kata both with iteration and recursion.

Contexts to use this Kata

The first part of this Kata is about parsing, and there are other Katas that also do so, for example Minesweeper, Args. The other parts are mostly about making your calculation code clear and concise, and reflect the domain language of the problem.

Kata: Bowling

Create a program, which, given a valid sequence of rolls for one line of American Ten-Pin Bowling, produces the total score for the game. This is a summary of the rules of the game:

- Each game, or "line" of bowling, includes ten turns, or "frames" for the bowler.
- In each frame, the bowler gets up to two tries to knock down all the pins.
- If in two tries, he fails to knock them all down, his score for that frame is the total number of pins knocked down in his two tries.
- If in two tries he knocks them all down, this is called a "spare" and his score for the frame is ten plus the number of pins knocked down on his next throw (in his next turn).
- If on his first try in the frame he knocks down all the pins, this is called a "strike". His turn is over, and his score for the frame is ten plus the simple total of the pins knocked down in his next two rolls.
- If he gets a spare or strike in the last (tenth) frame, the bowler gets to throw one or two more bonus balls, respectively. - These bonus throws are taken as part of the same turn. If the bonus throws knock down all the pins, the process does not repeat: the bonus throws are only used to calculate the score of the final frame.
- The game score is the total of all frame scores.

Here are some things that the program will not do:

- We will not check for valid rolls.
- We will not check for correct number of rolls and frames.
- We will not provide scores for intermediate frames.

The input is a scorecard from a finished bowling game, where "X" stands for a strike, "-" for no pins bowled, and "/" means a spare. Otherwise figures 1-9 indicate how many pins were knocked down in that throw.

Sample games:

```
12345123451234512345
```

always hitting pins without getting spares or strikes, a total score of 60

```
XXXXXXXXXXXX
```

a perfect game, 12 strikes, giving a score of 300

```
9-9-9-9-9-9-9-9-9-9-
```

heartbreak - 9 pins down each round, giving a score of 90

```
5/5/5/5/5/5/5/5/5/5/5
```

a spare every round, giving a score of 150

Additional discussion points for the Retrospective

- Did you do any design before you started coding? If so, does your code have this design now?
- At what point did you realize you can't simply loop over frames, that you in fact need to refer to the previous frame as well as the current one in order to calculate the score? In an ideal world, when should you have realized this?
- Look at the code you have ended up with, and compare it with the above description of the rules of bowling. Is there any similarity in the words used in each?
- Did you do enough refactoring? How would you know?
- How did you decide which tests to write, and in which order? Did it matter what order you implemented them in?

Ideas for after the dojo

Read this article "Engineer Notebook: An Extreme Programming Episode"[17] by Robert C. Martin, where he describes solving this kata together with Robert S. Koss. Follow along in your editor. Does he do the kata the same way as you would?

Contexts to use this kata

This kata is relatively easy for newcomers to TDD. It's a good one for creating a test list at the start, and choosing a suitable order to implement them in that allows the algorithm to develop organically.

As Robert Martin points out in his article, it is possible to analyse the requirements for this problem and come up with an over-designed solution. If you do this Kata with TDD newcomers it could show them how TDD can help you discover a good design for your software without too much up-front work.

[17]http://www.objectmentor.com/resources/articles/xpepisode.htm

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?
- Does your code follow the Open-Closed Principle?
- What if a new requirement came along that multiples of seven were “Whizz”?

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.

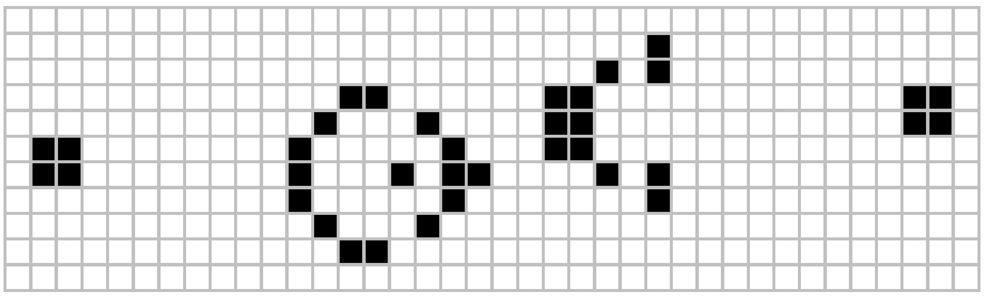
Kata: Game Of Life

The Game of Life is a cellular automaton devised by the British mathematician John Horton Conway in 1970. It's a zero-player game, meaning that its evolution is determined by its initial state, requiring no further input - you create an initial configuration and watch how it evolves. Some initial patterns give rise to complex and beautiful ever-changing scenarios, others eventually stabilize into a fixed configuration.

The universe of the Game of Life is an infinite two-dimensional orthogonal grid of square cells, each of which is in one of two possible states, alive or dead. Every cell interacts with its eight neighbours, which are the cells that are horizontally, vertically, or diagonally adjacent. At each step in time, the following transitions occur:

- Any live cell with fewer than two live neighbours dies, as if caused by under-population.
- Any live cell with two or three live neighbours lives on to the next generation.
- Any live cell with more than three live neighbours dies, as if by overcrowding.
- Any dead cell with exactly three live neighbours becomes a live cell, as if by reproduction.

The initial pattern constitutes the seed of the system. The first generation is created by applying the above rules simultaneously to every cell in the seed — births and deaths occur simultaneously, and the discrete moment at which this happens is sometimes called a tick (in other words, each generation is a pure function of the preceding one). The rules continue to be applied repeatedly to create further generations.



Gosper Glider Gun (image from wikipedia)

This aim of this Kata is to write code that takes a seed position, and calculates the following generations. It's also useful to write code to visualize the cells as the position changes after each new generation.

Additional discussion points for the Retrospective

- Did you start with a "Cell" class or a "World" class? Or the "tick" function? How did that initial decision affect the code you ended up with?

- Did you find a good datastructure to represent an infinite two-dimensional orthogonal grid?
- What would happen if the requirements changed now and instead of two-dimensional, the grid became three-dimensional? Is the knowledge about x, y coordinates spread all over your code and in all the method signatures?
- How well does your code reflect the domain language of the problem? For example do you have the concept of a "tick" or a "seed" in your code?

Ideas for after the dojo

This is the most popular kata done at Code Retreat events, and a lot of people have done it. There are lots of screencasts and blog posts about it that you could look at.

If you do the kata again, try starting from a different place. If you started with a "Cell" class, try starting with a "tick" function. Or try starting with the visualization of the next generations. Or with parsing an ASCII art representation of the seed. Or create a clickable grid for people to input their seed. There are lots of aspects to this Kata, and in one dojo you won't have explored all the possibilities.

Contexts to use this Kata

This is a good Kata for a functional programming style. It's also good for thinking about using domain language in code. There are other katas that use a two dimensional grid, you might find it interesting to compare this one with Minesweeper and Reversi.

Kata: Gilded Rose[18]

Hi and welcome to team Gilded Rose. As you know, we are a small inn with a prime location in a prominent city ran by a friendly innkeeper named Allison. We also buy and sell only the finest goods. Unfortunately, our goods are constantly degrading in quality as they approach their sell by date. We have a system in place that updates our inventory for us. It was developed by a no-nonsense type named Leeroy, who has moved on to new adventures. Your task is to add the new feature to our system so that we can begin selling a new category of items. First an introduction to our system:

- All items have a SellIn value which denotes the number of days we have to sell the item
- All items have a Quality value which denotes how valuable the item is
- At the end of each day our system lowers both values for every item

Pretty simple, right? Well this is where it gets interesting:

- Once the sell by date has passed, Quality degrades twice as fast
- The Quality of an item is never negative
- "Aged Brie" actually increases in Quality the older it gets
- The Quality of an item is never more than 50
- "Sulfuras", being a legendary item, never has to be sold or decreases in Quality
- "Backstage passes", like aged brie, increases in Quality as it's SellIn value approaches; Quality increases by 2 when there are 10 days or less and by 3 when there are 5 days or less but Quality drops to 0 after the concert

We have recently signed a supplier of conjured items. This requires an update to our system:

- "Conjured" items degrade in Quality twice as fast as normal items

Feel free to make any changes to the UpdateQuality method and add any new code as long as everything still works correctly. However, do not alter the Item class or Items property as those belong to the goblin in the corner who will insta-rage and one-shot you as he doesn't believe in shared code ownership (you can make the UpdateQuality method and Items property static if you like, we'll cover for you).

Just for clarification, an item can never have its Quality increase above 50, however "Sulfuras" is a legendary item and as such its Quality is 80 and it never alters.

[18]With thanks to Terry Hughes, who designed this Kata, for permission to include it here.

The Code

You can download the code for this system from my repo on GitHub[19]. There are versions there in many programming languages, including C#, Java, Python, Ruby, Smalltalk, C, C++. Pick whichever one you prefer, they are all translations from the original C# version.

Two approaches to this Kata

This is designed as a refactoring kata, where you take this less than clean code, and transform it via small steps into something that can be maintained and extended. When you have the code under control, it should be easy to add the new feature for “Conjured” items. In the original version of this Kata, there were no tests provided, only a textual description of the requirements, (above). In a later addition, I added Text-Based (aka Approval) tests. So you can do this kata in two ways, either writing your own tests, and practice writing really good ones, or just jump straight to the refactoring part, leaning on the text-based tests.

In either case I recommend that while you’re doing the refactoring, that you use a Kata visualization tool like Cyber-Dojo or the CodersDojo client, so you can review how large steps you took. The aim is for as small as possible, without any refactoring mistakes that cause unexpected test failures.

Kata approach #1: Practice designing test cases

Ignore the text-based tests for the code, and concentrate on writing your own tests. The aim is to design good, readable tests that you can lean on when you start refactoring. As far as your test tools allow, use the domain language of the problem to express the tests.

If you feel you already have good skills at designing test cases, you could add an additional challenge by using an unfamiliar testing framework. Perhaps divide the dojo participants into pairs, and have each trying out a different tool. For example a unit testing framwork like JUnit[20], a BDD framework like RSpec[21], and a functional test framework like Cucumber[22] or Fitnesse[23]. You could also have a pair using the text-based tests provided, (see next section).

Kata approach #2: Use the provided Text-Based tests

Instead of writing all the tests yourself, you could try using the text-based tests, (found in the repo in the subfolder “texttests”). This lets you concentrate on the Refactoring part of the Kata. I’ve provided test fixtures that let you run the same tests against each language version of the kata. The text-based tests give you full branch and statement coverage of the code, and can be

[19]https://github.com/emilybache/Refactoring-Katas/tree/master/GildedRose
[20]http://junit.org
[21]http://rspec.info/
[22]http://cukes.info/
[23]http://fitnesse.org/

conveniently run using the open source tool TextTest[24]. There is more information about how to do this in the README file in the "texttests" folder.

Additional discussion points for the Retrospective

- Did you make mistakes while refactoring that the tests caught?
- Did you miss important tests that you had to add later?
- Did you obey the instructions about not modifying the Item class? How did that affect your design options?
- Review your session in the Kata visualization tool, (if you used one). How large steps did you take? Could you have taken smaller steps?
- When you were refactoring, did you keep in mind the new feature you intended to add? Did that affect your design choices?
- Think about the design you ended up for the code. How easy would it be to add new classes of items? Produce weekly reports detailing current inventory and overall quality? Add functionality for updating quality values when stocktaking, (comparing what's on the shelves with what's in the books)?

If you wrote your own tests

- Have you tried running the text-based tests to see if they catch errors your tests miss?
- Did you find any bugs in the code when writing your tests?
- How good is the statement coverage of your test code?

If you used the Text-Based tests

- Were the tests good enough to support your refactoring efforts?
- Could you easily identify the cause of failing tests?
- How does working with these kind of tests differ from working with unit tests?

[24]http://texttest.org

Ideas for after the dojo

Watch this screencast[25] of Bobby Johnson tackling this Kata, and review the test cases[26] that he created (in C#). How do they compare with your test cases, either the ones you designed or the text-based ones? How do they differ in terms of coverage, readability, robustness and speed? (You can read more about these desirable properties of test cases in the chapter "Principles for Agile Automated Test Design")

Contexts to use this Kata

This is a good Kata for practicing your refactoring skills, but you can also use it to learn to design better test cases, and to try out a text-based testing approach. It would be ambitious to do all three in the same dojo meeting however.

[25]http://vimeo.com/34091297

[26]https://github.com/NotMyself/GildedRose/blob/first_refactor/src/GildedRose.Tests/UpdateItemsTests.cs

Kata: Medicine Clash

As a Health Insurer,

I want to be able to search for patients who have a medicine clash,

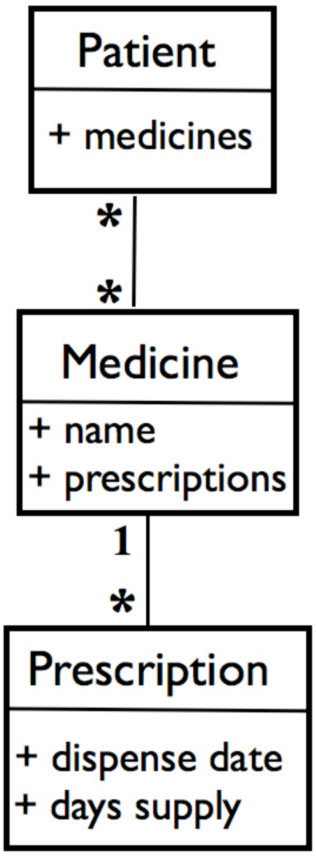
So that I can alert their doctors and get their prescriptions changed.

Health Insurance companies don't always get such good press, but in this case, they actually do have your best interests at heart. Some medicines interact in unfortunate ways when they get into your body at the same time, and your doctor isn't always alert enough to spot the clash when writing your prescriptions. Often, medicine interactions are only identified years after the medicines become widely used, and your doctor might not be completely up to date. Your Health Insurer certainly wants you to stay healthy, so discovering a customers has a medicine clash and getting it corrected is good for business, and good for you!

For this Kata, your task is to search a database of patient records and find the ones that have recently been taking prescription medicines that clash. For each patient, look at the most recent 90 days, and calculate the number of days they have taken all of a list of medicines that are known to clash.

Data Format

You can assume the data is in a database, which is accessed in the code via an object oriented domain model. The domain model is large and complex, but for this problem you can ignore all but the following entities and attributes:



Entities and attributes for the Medicine Clash Kata

In words, this shows that each Patient has a list of Medicines. Medicines have a unique name. Each Medicine has a list of Prescriptions. Each Prescription has a dispense date and a number of days supply.

Alternatively the data could be in a .csv file with column headings like this:

```
patient id, medicine name, prescription, dispense date, days supply
```

There would be one row in the file for each prescription, ie several rows for each patient. The "dispense date" could be listed in ISO format to make it easy to parse, eg 2012-11-01.

You can assume:

- patients start taking the medicine on the dispense date.
- the "days supply" tells you how many days they continue to take the medicine after the dispense date.
- if they have two overlapping prescriptions for the same medicine, they stop taking the earlier one. Imagine they have mislaid the medicine they got from the first prescription when they start on the second prescription.

Additional discussion points for the Retrospective

- How did you handle the fact that the code relies on the current date? Did you use a mock or a stub?
- Did you learn how to use your date library better when you were doing this kata?
- How did you plan the test cases for this kata? Did you try to be comprehensive for all conceivable kinds of date overlap relative to the current date?
- Have you used set operations like "intersection" and "union" in your code? If not, would doing so make your code more readable?

Ideas for after the dojo

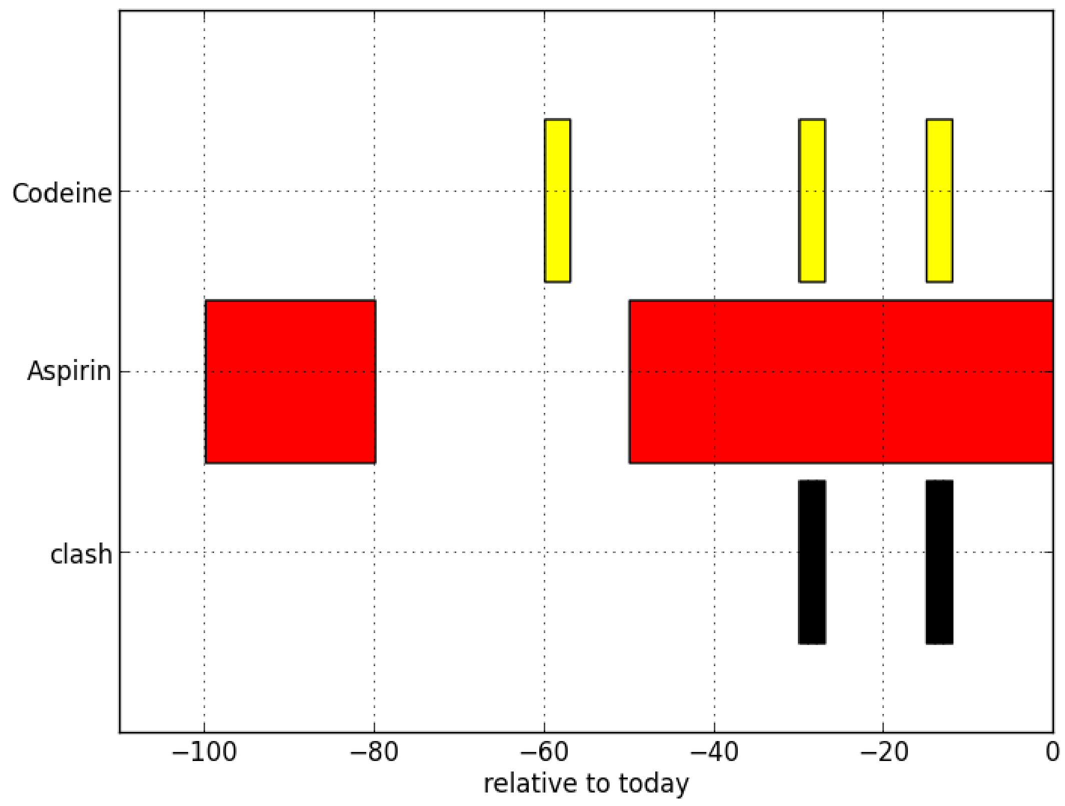
When you've had a go at the problem yourself, and think you have a good solution for it, you might be interested to review the sample solution on my github page[27], with code in Python and Ruby. Read the code and tests and think about:

- Can you explain the algorithm that has been chosen to solve the problem?
- How readable are the tests? Do you understand what is being tested?

I find this sample code fascinating. It follows clean code guidelines with well named, short methods, and has lots of unit tests. I also find it very hard to understand. What do you think?

Another thing you could do is write some code that draws a graph visualizing when patients have taken each medicine. Something like this:

[27] https://github.com/emilybache/KataMedicineClash



Sample Medicine Clash Visualization

Does this kind of visualization help you to identify all the edge cases you need to cover in your tests?

Contexts to use this Kata

This is a good Kata for learning about how to test code that depends on the current date, and for practicing using dates generally. It also lends itself to a functional style of programming. This is a good Kata for building up an algorithm gradually, and ensuring you have tests for all the important edge cases.

Kata: Minesweeper [28]

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 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:

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

```
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.

Kata: Monty Hall

This eponymous gameshow host tempts contestants with a big prize, hidden behind one of three doors. The contestant begins by choosing a door, but not opening it. Then Monty steps forward and opens one of the other doors. He reveals a goat (!). Then the contestant has the choice of either sticking with the door they have already chosen, or switching to the other unopened door. Whichever door the contestant decides on will be opened, and if they find the prize, they get to keep it. (I'm not sure what happens if they get the second goat!) So what's the best strategy?

The Monty Hall Dilemma

That's the classic "Monty Hall Dilemma". Should you keep to your original choice of door? You could try playing the game yourself using this online version[29]?[30]

People are biased towards sticking with what they've chosen, and the vast majority of people keep the door they originally picked. Intuitively there should be an equal chance of the prize being behind any of the three doors, so it shouldn't matter if you stick or switch. However, in this case, your intuition is wrong. You are twice as likely to win the prize if you switch to the other unopened door.

I was not the only one to disbelieve this result, apparently even famous mathematicians have refused to accept it. What finally convinced them, was a computer simulation. So, your task is to write that computer simulation. Prove that switching doors is best, by simulating 1000 games using each strategy and comparing the winning percentage.

Additional discussion points for the Retrospective

- How did you set up your "guiding test" at the start? Did the interface you designed turn out to be a good one?
- How did you allow the tests to control the randomness of which door would have the prize?
- Did you use any mocks or stubs?
- How easily would your code cope if the number of doors were increased?

Ideas for after the dojo

- Read this essay[31] by Martin Fowler about mocks and stubs and see if you can do this kata using either style of TDD.

[29]http://redsquirrel.com/dave/play/montyOnAjax.html

[30]This version was designed & built by Dave Hoover. Additional benefit: you don't run the risk of winning an actual goat.

[31]http://martinfowler.com/articles/mocksArentStubs.html

Contexts to use this kata

This kata is a little bit challenging to break down into test cases you can implement in small steps. You also have to understand how to isolate the random elements and make the code testable. Before you do the kata in your dojo, you might want to go through the theory of the difference between mocks and stubs. You might find it easier to do this kata with a mockist approach (London School) than a classic approach.

Kata: Reversi

Reversi is a board game for two players, using a 8x8 two-dimensional grid of square cells. The players take it in turns to place a disc showing their colour into a square, and take control of one or more of the discs showing the opponent's colour. The game ends when either all 64 squares are filled with a disc, or there are no legal positions for either player to place a disc. The winner is the player with the most discs showing their colour on the board.

A position is legal if it allows you to flip over one or more of the opponent's discs, ie change them to your colour. For this to happen you must place the new disc in a position where there is at least one straight (horizontal, vertical, or diagonal) occupied line between the new disc and another disc of your colour, with one or more contiguous pieces of the opponent's colour between them. For example if we have a board like this:

```
. . . . . . . .
. . . . . . . .
. . . . . . . .
. . . W B . . .
. . . B W . . .
. . . . . . . .
. . . . . . . .
. . . . . . . .
```

(where "." indicates an empty square, "B" indicates a black disc, and "W" a white disc)

there are legal moves for Black in the squares marked with a *:

```
. . . . . . . .
. . . . . . . .
. . . * . . . .
. . * W B . . .
. . . B W * . .
. . . . * . . .
. . . . . . . .
. . . . . . . .
```

Note there is more detailed information about the rules on Wikipedia[32].

For this Code Kata, you write a program that takes an arbitrary position and whose turn it is, and find all the legal positions where they could play. You don't have to use ASCII art in your code, you could output the legal positions as co-ordinates, (columns labelled A - H, rows labelled 1 - 8 starting from top left hand corner), like this: [C4, D3, E6, F5]. Alternatively come up with a better graphical visualization of the position.

[32]http://en.wikipedia.org/wiki/Reversi

Additional discussion points for the Retrospective

- Did you find a good datastructure to represent the two-dimensional orthogonal grid in your code?
- Did you design test cases that were small enough to get you started, yet were still passing at the end of the Kata?
- Would your code still work if the size or shape of the board changed?

Ideas for after the dojo

If you used an object oriented language, try it again in a functional language, and vice versa.

Contexts to use this Kata

There are other katas that use a two dimensional grid, you might find it interesting to compare this one with Minesweeper and Game of Life.

Four Katas on a SOLID Theme

Imagine you have just started a new job working on the software systems for a Formula 1 racing team. You have inherited some code that you would characterize as legacy, and you now want to write unit tests for it.

Clone the repository on github[33], to find starting code for these four Katas that form a set. The starting code is available in a variety of programming languages, including C#, Java, Javascript and Python. Note that these katas were originally designed by Luca Minudel and published here on github[34] as part of his research into TDD and Design Principles. Also note - he really has worked for a Formula 1 racing team!

It's rather ambitious to do all four exercises in one dojo meeting. I suggest doing them over two or three meetings. I've listed the exercises here in the order you could do them in.

Kata: Tyre Pressure Monitor[35]

The code for this Kata is in the folder called "**TirePressureMonitoringSystem**".

The Alarm class is designed to monitor tyre pressure and set an alarm if the pressure falls outside of the expected range. The Sensor class provided for the exercise fakes the behaviour of a real tyre sensor from a racing car, providing random but realistic values.

- Write the unit tests for the **Alarm** class, refactor the code as much as you need to make the class testable. Don't change any interfaces that other code (not included here) may rely on.

Kata: Unicode File to HTML Text Converter

The code for this Kata is in the folder called "**UnicodeFileToHtmTextConverter**".

The UnicodeFileToHtmlTextConverter class is designed to reformat a plain text file for display in a browser.

- Write the unit tests for the **UnicodeFileToHtmlTextConverter** class, refactor the code as much as you need to make the class testable. Don't change any interfaces that other code (not included here) may rely on.

[33]https://github.com/emilybache/TDDwithMockObjectsAndDesignPrinciples/tree/master/TDDMicroExercises

[34]https://github.com/lucaminudel/TDDwithMockObjectsAndDesignPrinciples

[35]The American spelling of "Tyre" is "Tire".

Kata: Turn Ticket Dispenser

The code for this Kata is in the folder called “**TurnTicketDispenser**”.

The TicketDispenser class is designed to be used to manage a queuing system in a shop. There may be more than one ticket dispenser but the same ticket should not be issued to two different customers.

- Write the unit tests for the **TicketDispenser** class, refactor the code as much as you need to make the class testable. Don’t change any interfaces that other code (not included here) may rely on.

Kata: Racing Car Telemetry

The code for this Kata is in the folder called “**TelemetrySystem**”.

The responsibility of the TelemetryDiagnosticControls class is to establish a connection to the telemetry server (through the TelemetryClient), send a diagnostic request and successfully receive the response that contains the diagnostic info. The TelemetryClient class provided for the exercise fakes the behavior of the real TelemetryClient class, and can respond with either the diagnostic information or a random sequence. The real TelemetryClient class would connect and communicate with the telemetry server via tcp/ip, and receive information from the racing car.

- Write the unit tests for the **TelemetryDiagnosticControls** class, refactor the code as much as you need to make the class testable. Don’t change any interfaces that other code (not included here) may rely on.

Additional discussion points for the Retrospective

- What do you think of the design of the code you ended up with? Is it clean code?
- What was it about this code that made it hard to write tests for without refactoring?
- What SOLID principle(s) did this code violate before you started? Does it still violate any?
- Were there any other problems with this code that are not to do with SOLID violations? Did your testing approach lead you to find them, or did you notice them some other way?

Ideas for after the dojo

Re-read the descriptions of Robert C. Martin’s SOLID principles and try to find violations of them in the production code you’re working on. If you found other problems in the Kata code that aren’t to do with SOLID violations, look for them too. How has your testing approach allowed these problems to be there? Should you change your testing approach?

Contexts to use this kata

Before you do these katas in your dojo, you will want to remind people of the SOLID principles, to put the katas in context. For an explanation of SOLID see wikipedia[36], or the book "Agile Software Development: Principles, Patterns, and Practices" by Robert C. Martin.

All these Katas are interesting to do with a London School approach to TDD. Luca's original study was looking at whether this leads to better adherence to SOLID principles than using classic TDD. In one of the exercises, "Tyre Pressure Monitor", there is some test code provided - a hand-made Mock and Stub implementation for the Alarm interface. This code is not needed to complete the exercises, it's just there to help you to understand what a Mock and a Stub are. You could go through this code at the start of the dojo, to help people just beginning to learn the London School of TDD.

[36]http://en.wikipedia.org/wiki/SOLID_%28object-oriented_design%29

Kata: String Calculator[37]

Before you start

- Try not to read ahead.
- Do one task at a time. The trick is to learn to work incrementally.
- There is no need to test for invalid inputs for this kata, assume the string you receive is correctly formatted.

1. Create a simple String calculator with a method **int Add(string numbers)**
    - The method can take 0, 1 or 2 numbers, and will return their sum (for an empty string it will return 0) for example "" or "1" or "1,2"
    - Start with the simplest test case of an empty string and move to one and two numbers
    - Remember to solve things as simply as possible
    - Remember to refactor after each passing test
2. Allow the Add method to handle an unknown amount of numbers
3. Allow the Add method to handle newlines between numbers instead of commas.
    - The following input is valid: "1\n2,3" (will equal 6)
    - The following input is **not** valid: "1,\n" (no need to handle this in your code)
4. Support different delimiters
    - To change a delimiter, the beginning of the string will contain a separate line that looks like this: "//[delimiter]\n[numbers...]" for example "//;\n1;2" should return three since the delimiter is ';' .
    - The first line is optional, so all existing scenarios should still be supported, (existing tests should still pass).
5. Calling Add with a negative number should throw an exception "negatives not allowed". The exception message should include the negative that was passed. If there are multiple negatives, list all of them in the message.

[37]With thanks to Roy Osherove, who designed this Kata, for permission to include it here.

Additional discussion points for the Retrospective

- Did you ever write more code than you needed to make the current tests pass?
- Did you ever have more than one failing test at a time?
- Did the tests fail unexpectedly at any point? If so, why?
- How much did writing the tests slow you down?
- Did you write more tests than you would have if you had coded first and written tests afterwards?
- Are you happy with the design of the code you ended up with? Should you have refactored it more often?

Ideas for after the dojo

Watch some of the example screencasts on Roy Osherove's website[38]. Many good programmers have contributed there. Is there a screencast in the language and tools you're using? How does it change the way you do TDD if you use a different programming language or toolset?

Contexts to use this kata

This is an excellent Kata for TDD beginners. It leads you through some simple test cases and code, making you follow the rhythm of red-green-refactor. Experienced TDD:ers can profitably use it to try out new tools or languages.

[38]http://osherove.com/tdd-kata-1/

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 "player1Scores()" being called on your class, and similarly player2Scores() for the other button. Afterwards, you will get a call "currentScore()" from the scoreboard asking what it should display. This method should return a string with the current score.

You can read more about Tennis scores here[39] 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 the opponent.
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" for the player in the lead.

You need only report the score for the current game. Sets and Matches are out of scope.

Refactoring version

Note that this can be done as a refactoring kata, where you take some less than clean code, and transform it via small steps into something more maintainable. What is nice about this Kata is that it is not too hard to have (virtually) exhaustive tests so any mistakes you make while refactoring should be very obvious. There are three versions of refactorable code with tests available on github[40], 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 [41], so you can review how large steps you took. The aim is for as small as possible, with as few refactoring mistakes as possible.

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

[40]https://github.com/emilybache/Refactoring-Katas/tree/master/Tennis

[41]See the chapter (#ToolsForTheDojo)

Additional discussion points for the Retrospective

- Is the code you have ended up with clean? Are there any smells?
- Did you make mistakes while refactoring that were caught by the tests?
- Are your tests exhaustive?
- If you did this as a refactoring kata, and used a tool to record your progress, review it. Could you have taken smaller steps?

Ideas for after the dojo

- If you did this as a normal kata, try it as a refactoring kata (code on github[42])
- 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.

[42]https://github.com/emilybache/Refactoring-Katas/tree/master/Tennis

Kata: Trivia

I’d like to get permission from J.B. Rainsberger to include his kata here.

Kata: Yahtzee[43]

The game of Yahtzee is a simple dice game. Each player rolls five six-sided dice. They can re-roll some or all of the dice up to three times (including the original roll).

For example, suppose a players rolls (3,4,5,5,2). They hold (-,-,5,5,-) and re-roll (3,4,-,-,2) to get (5,1,5,5,3). They decide to hold (5,-,5,5,-) and re-roll (-,1,-,-,3). They end up with (5,6,5,5,2).

The player then places the roll in a category, such as ones, twos, fives, pair, two pairs etc (see below). If the roll is compatible with the category, the player gets a score for the roll according to the rules. If the roll is not compatible with the category, the player scores zero for the roll.

For example, suppose a player scores (5,6,5,5,2) in the fives category they would score 15 (three fives). The score for that go is then added to their total and the category cannot be used again in the remaining goes for that game. A full game consists of one go for each category. Thus, for their last go in a game, a player must choose their only remaining category.

Your task is to score a *given* roll in a *given* category. You do **not** have to program the random dice rolling. The game is **not** played by letting the computer choose the highest scoring category for a given roll.

Categories and Scoring Rules

Note that the rules below differ from the original (copyrighted) rules.

Chance: The player scores the sum of all dice, no matter what they read. For example,

- 1,1,3,3,6 placed on "chance" scores 14 (1+1+3+3+6)
- 4,5,5,6,1 placed on "chance" scores 21 (4+5+5+6+1)

Yahtzee: If all dice have the same number, the player scores 50 points. For example,

- 1,1,1,1,1 placed on "yahtzee" scores 50
- 1,1,1,2,1 placed on "yahtzee" scores 0

Ones, **Twos**, **Threes**, **Fours**, **Fives**, **Sixes**: The player scores the sum of the dice that reads one, two, three, four, five or six, respectively. For example,

- 1,1,2,4,4 placed on "fours" scores 8 (4+4)
- 2,3,2,5,1 placed on "twos" scores 4 (2+2)
- 3,3,3,4,5 placed on "ones" scores 0

[43]With thanks to Jon Jagger, who designed this Kata, for permission to include it here.

Pair: The player scores the sum of the two highest matching dice. For example, when placed on "pair"

- 3,3,3,4,4 scores 8 (4+4)
- 1,1,6,2,6 scores 12 (6+6)
- 3,3,3,4,1 scores 0
- 3,3,3,3,1 scores 0

Two pairs: If there are two pairs of dice with the same number, the player scores the sum of these dice. For example, when placed on "two pairs"

- 1,1,2,3,3 scores 8 (1+1+3+3)
- 1,1,2,3,4 scores 0
- 1,1,2,2,2 scores 0

Three of a kind: If there are three dice with the same number, the player scores the sum of these dice. For example, when placed on "three of a kind"

- 3,3,3,4,5 scores 9 (3+3+3)
- 3,3,4,5,6 scores 0
- 3,3,3,3,1 scores 0

Four of a kind: If there are four dice with the same number, the player scores the sum of these dice. For example, when placed on "four of a kind"

- 2,2,2,2,5 scores 8 (2+2+2+2)
- 2,2,2,5,5 scores 0
- 2,2,2,2,2 scores 0

Small straight: When placed on "small straight", if the dice read (1,2,3,4,5), the player scores 15 (the sum of all the dice).

Large straight: When placed on "large straight", if the dice read (2,3,4,5,6), the player scores 20 (the sum of all the dice).

Full house: If the dice are two of a kind and three of a kind, the player scores the sum of all the dice. For example, when placed on "full house"

- 1,1,2,2,2 scores 8 (1+1+2+2+2)
- 2,2,3,3,4 scores 0
- 4,4,4,4,4 scores 0

Refactoring Kata

Jon Jagger has designed a refactoring version of this kata. I've taken the liberty of putting various language versions of it here on github[44], or you can do the kata directly in the Cyber-Dojo. See the article "Yahtzee Cyber-Dojo Refactoring"[45] for information about that. While you are doing this refactoring kata, note down all the code smells you see and address.

Additional discussion points for the Retrospective

- How much duplication is there in your solution? In your test code?
- Did you write a list of test cases before you started? How did you decide what order to implement them in?
- If you did this as a refactoring kata, discuss the code smells you identified. Do you have them in your production code?

Ideas for after the dojo

Do this kata again from scratch and tackle the test cases in a different order. Does this affect the design you end up with? Can you take the tests in any order at all?

If you'd like more practice at choosing test case order, try the Minesweeper Kata. Be sure to make a list of test cases before you start, and be mindful of what order you could best implement them in.

Contexts to use this kata

This Kata is quite easy for TDD beginners since the test cases are more or less enumerated in the problem description. The order to implement them in is not prescribed, though, so you can practice that aspect of TDD.

[44]https://github.com/emilybache/Refactoring-Katas/tree/master/Yahtzee

[45]http://jonjagger.blogspot.co.uk/2012/05/yahtzee-cyber-dojo-refactoring-in-java.html

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

London School of TDD

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

Interesting Books for Coders (except with less actual code)

- "Extreme Programming Explained", Kent Beck (I recommend the first edition if you can get hold of it.)
- "The Pragmatic Programmer", Andrew Hunt and David Thomas
- "Apprenticeship Patterns: Guidance for the Aspiring Software Craftsman", Dave Hoover, Adewale Oshineye