Gentleman Programming Book

> A clean programmer is the best kind of programmer
>
> *- by Alan Buscaglia*

Chapter 1

Clean Agile

- Problems of waterfall
- Why agile?
- Why you think you are doing agile but in reality...you don't
- Extreme Programming
- TDD
- Atomic design, Front End point of view
- Functional Programming
- User Story and TDD

Chapter 2

Communication First and Foremost

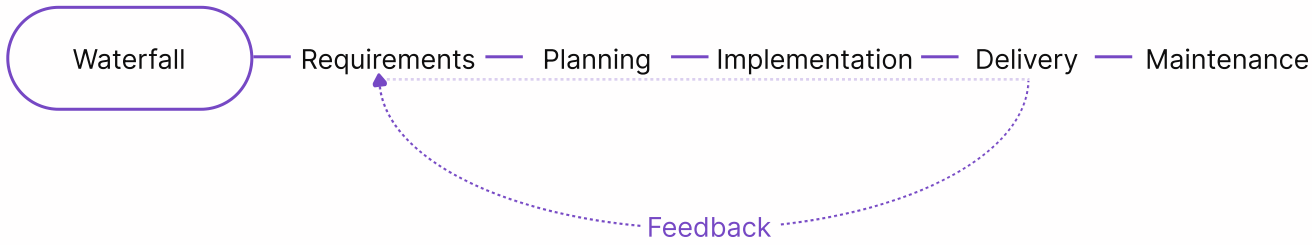
- Companies are not grounded to a certain location
- Coordinating across time zones
- There are some other easy ways we improve this mindset
- Creating Rapport

Clean Agile

Fantastic! Everything is agile today, all companies love agile, the entire world does agile, but ...are they?

Problems of waterfall:

Let's talk about waterfall, yeah, the bad boy in town, the one everybody hates. Waterfall in The main idea is:



- We get the requirements, what we want to do, the stakeholders needs.
- You plan how to do it, often comes with an analysis and design of the solution.
- You implement it, creating working software.
- You deliver the software and wait for feedback, creating documentation during the process.
- Maintenance of the working solution.

Once we deliver the solution, we ask for feedback and if we need to touch anything, we start the process all over again.

This is awesome as long as the requirements are super stable and we know they will not change during implementation, something that in the real world is practically impossible as they are ALWAYS changing.

If we were a fabric we wouldn't have any of these issues, as we know the specific required materials to create something, we put them in the machine, and the result is always going to be the same.

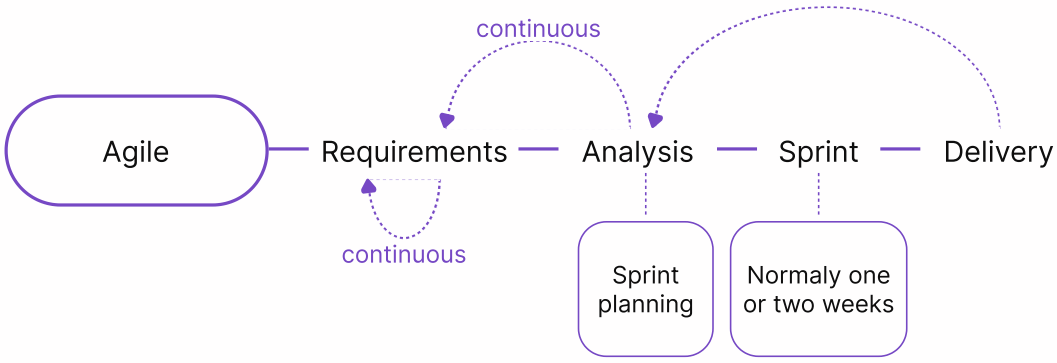
But we work with software solutions to meet people's needs, and these are always changing, evolving. Here comes the biggest problem with the waterfall methodology. We have to wait until the end of the implementation to receive feedback and then start the whole process again, so what if the user needs have changed in the meantime ? We've just wasted a lot of useful time.

One great analogy is the airplane's pilot one, you would like to be informed as soon as possible if there's any problem with the plane and not wait until the engine fails, or even worse, the airplane crashes to receive a notification.

Why agile?

Now here's the thing, a project it's an always evolving succession of events, for example, the analysis never ends ! so getting feedback as soon as possible is the main key of the Agile methodology. Here, we search for the stakeholders involvement in the whole process, delivering minimal amounts of functionalities waiting for a, let's hope positive, response; and if it's a negative one, no problems at all, we can attack them as soon as possible without having to wait for the end of the world to do so.

So how companies usually use agile is the following:



- We continuously get requirements, as we work on small functionalities we can choose what are the most critical needs and implement a plan of action to deliver small parts that can fulfill this end.
- We continuously do an analysis of the requirements to prepare future work. The amount will correspond to the timeframe we already decided according to the needs.
- Now with everything prepared we are comfortable to start working on our tasks inside a sprint. It represents the time frame we decided in which we compromise ourselves to deliver a certain amount of work and it can be variable depending on the need.
- We deliver the functionalities and continue with the process again. The main difference is that we start the work with stakeholders feedback from the previous iteration.

We can fail to deliver inside an adjusted timeframe, and that's not a problem at first, as we measure the team and recollect feedback to adjust to the next sprint. After some iterations we can estimate the correct amount of work the team can deliver in a certain context.

Why you think you are doing agile but in reality...you don't

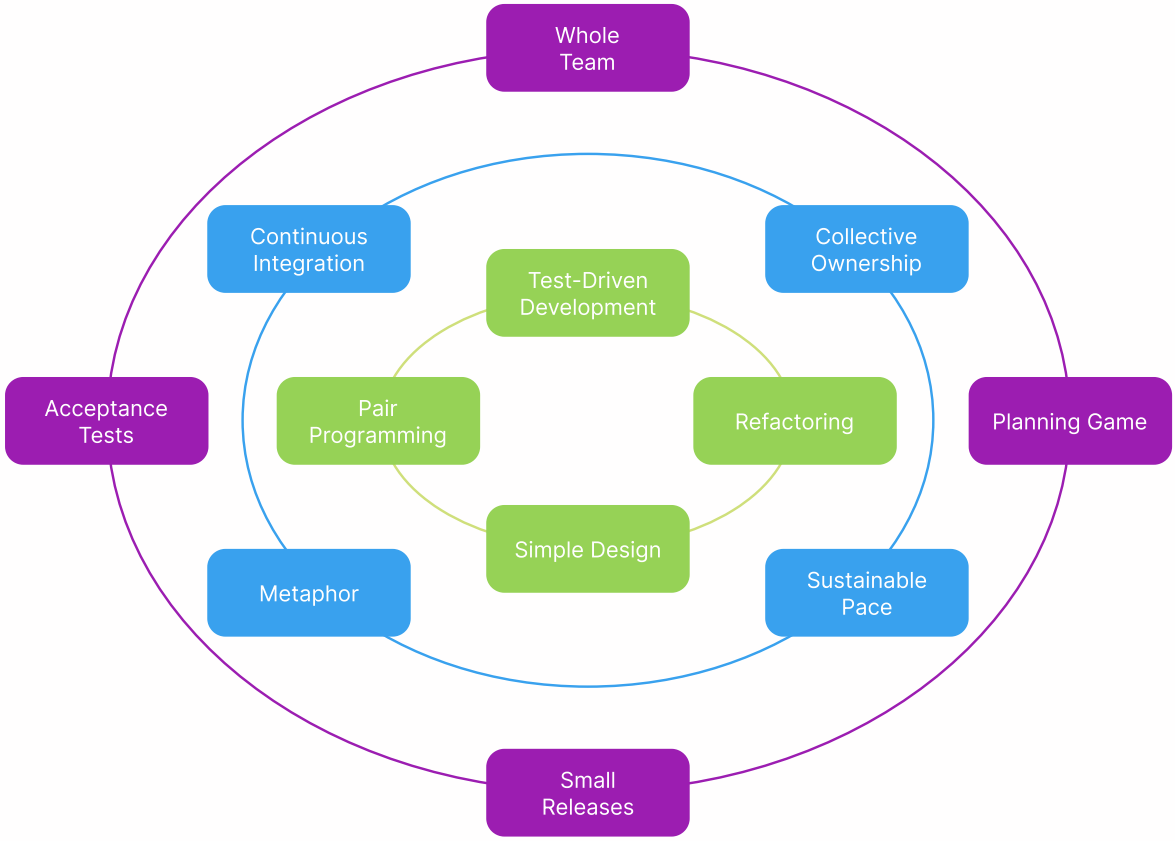
This is normal, you think you are doing agile because you have dailies and that makes the team agile, but in fact that is just one ceremony from many. You can't define being or not agile by the ceremonies that take place inside the project, as it's more of a way of thinking.

You may be working on sprints, using scrum, having retros, and all those amazing things but you also may be working in huge features, not delivering each sprint, not accepting any change until you finish your work or even being owner of your knowledge and not sharing it with the team. If you find yourself in any of these last items...ey...you are not doing agile.

Extreme Programming

This is an amazing practice that according to R.C Martin, co-founder of the agile manifesto and author of Clean Agile, is the true essence of it.

It consists of the organization of practices by three rings called the "Circle of Life". Each ring represents a different aspect of project development:



The "Outer Ring" represents the aspect of business, it contains all business focused practices which when put together creates the perfect environment for project development.

- **Planning Game:** grabbing the project and breaking it down into smaller pieces for better understanding and organization. Features, stories, tasks, etc.
- **Small Releases:** here comes what I was saying about attacking a whole functionality at once and how that could bring a waterfall approach to something that should be agile. We should always try to identify and prioritize the smaller pieces of value and work around them, being the work we want to deliver as soon as possible. The smaller the piece, the faster we will receive feedback and act accordingly.

- **Acceptance Tests:** now this is an easy one to understand but difficult to implement, we need to work in what we consider as a team as done, what are the requirements to really say something has been completely done, or at least on the agreed limitations. One recommendation is to think again in the minimum value we want to, and can, attack with the provided team. If we grab something really big it will be difficult to implement as we need to consider too many things, resulting in missed requirements, vague definitions and miscommunication. Consider creating functionalities that have a clear start and end, the result needs to be something that provides value on its own.
- **Whole team:** inside a proper project team, each member provides a certain functionality, we have our front-ends, back-ends, designers, product owners, project managers, etc. The main problem is always the same, how to communicate the work when it's so different and at the same time, dependent on each other (we will come back to this point in a little while).

The "Middle Ring" represents the aspect of the team, containing all team focused practices to improve team collaboration and interaction:

- **Sustainable Pace:** if you ask, when do you want this done ? the result will always be...well, as soon as possible ! and of course that's really difficult to do so, no because the team can't do it, most of them can, the problem is doing it every single time maintaining the same pace, it's impossible. Your team will be burned out at the third or fourth iteration and then no work will be done, the delivery speed will be greatly reduced. Again, think in small, contained functionalities, that can be delivered at a comfortable speed.
- **Collective Ownership:** how many times do you have to ask your pairs what the product owner is talking about in a meeting because you don't have the correct amount of context ? I'm not talking about those times you are gaming during the daily, but the vortex sucking all information that should be shared with the team and no one seems to know what's going on because no one ever told them. This is a super known issue in companies, the information happens at private quarters so only the people that were in the conversation know what's going on, and later on, try to communicate as best as possible the result with the rest of the team but they create a broken phone game in the process. The project needs to have a communication strategy to attack this kind of situation.
- **Continuous Integration:** as programmers we should be commiting code as much as possible, having the limit of not leaving a single day without new changes in the repository. There's always the possibility of working together in a functionality and missing communication efforts with each other. For example, one creates a method that does exactly the same as another, that was already created by a pair but didn't reach out to notify about it. Commiting changes as fast as possible will deliver feedback to your teammates and keep them updated to always be working on the "latest changes". If we wait to deliver new code after the functionality is done, we will enter waterfall territory all over again.
- **Metaphor:** If we will be working together in a project, we should all understand the context around in the same way, having definitions of each item. Having the same exact name to describe a certain item will bring a higher level of team understanding and leave out the confusion that could bring referring to the exact same item in more than one way. You could think of this as if each project is a different country, there are some of them that communicate in the exact same languages but they use different metaphors. For example, the United States and Londong both use English as their main language, but to represent being upset, American English uses "Disappointed" and British English uses "Gutted".

The "Inner Ring" represents the technical aspect, containing all practices related to improving technical work.

- **Pair Programming:** Sitting with each other to resolve a problem is not only going to make reaching a solution faster, but at the same time you are sharing your point of view between your pairs and also gaining theirs, with also the benefit of reaching a middle ground and creating a set of conventions that the team will follow after. Communication is key when working in a team, and having the possibility to work together to resolve a problem will bring feedback and context around the implementation.
- **Simple Design:** Here we go again, work small. We have already talked about this one but let's bring a little tip, let's say that we want to bring a certain functionality that represents a pretty big challenge to the team, we should always search for a way to provide the same amount of value by giving a much easier alternative. Sometimes we challenge ourselves and deliver a really complex but beautiful proposition of value, but the problem is that maybe that proposition goes nowhere because requirements change and we may find out that in reality the user doesn't want it, that's why also working as simple as possible is the way to go. You can always provide a simple but elegant solution to find the proper value and then iterate on something better.
- **Refactoring:** we all love the phrase "if it works, don't touch it", but that's not the correct mindset as we will enter in a spiral of legacy code by reusing no longer maintainable code. We need to refactor as much as possible. Technical debt is pretty much unavoidable, we always generate some bad quality code because of deadline's time constraints by implementing the fast, but not so correct, solution. One good way of dealing with it is to use part of the start or end of the sprint, according to the priority, to refactor the code, also this could be done using pair programming to use the benefits previously described.
- **Test-Driven Development:** We will talk about it later on, but we can define it as a process where we write our tests before even coding a single line. The main idea is that the requirements of the task define the tests we want to do and in result guide what we code.It can be a great ally when refactoring as we will understand later.

TDD

EVERYONE hates doing tests, for example clients hate PAYING companies for their Front End devs "wasting" time doing tests, and in the end... money. So why we, the wasters of time and money, should want to implement testing right?

Well, there are some things in my mind that can make up for all that hate:

- Code quality
- Code maintenance
- Coding Speed (yeah, you are reading this item correctly)

Understandable right ? you write tests so they pass the use cases, to write them you need to be organized because if not... it will be impossible to test your code. But there are things to consider, how do we write tests in a way that really increases the quality of our code in any meaningful way ?

First let's see what code quality means, and then I will tell you my take on what code quality means to ME.

If you look for an answer this is the one you may find:

"A quality code is one that is clear, simple, well tested, bug-free, refactored, documented, and performant"

Now, the measure of quality goes by the company requirements and the key points are usually reliability, maintainability, testability, portability, and reusability. It's really difficult to create code with 100% of quality, even Walter White couldn't create meth with more than 99.1% of purity; development problems, deadlines and other context and time consuming situations will arise endangering your code quality.

You can't write readable, maintainable, testable, portable, and reusable code if you are being rushed to finish a 4 point story task in just a morning (I really hope that's not your case, and if it is...you got this)

So here comes my take on what code quality is for me. Doing it, it's a mix of doing your best with the current tools, good practices and experience, against the existing context boundaries to create the cleanest code possible. My recommendation to all my students is to reach the objective first and then, if you have time, use it to improve the quality as high as possible. It is better to deliver an ugly thing than an incomplete, but beautiful, functionality. The quality of your code will increase with your experience along the way, as you gain more of it, you will know the best steps to reach an objective in the least amount of time and with the best practices.

Quality code also relates to the level of communication you can provide to your teammates or anyone in a simple glance. It's easy to see a code and say..wow, this is great ! and also say...wow, what a mess ! So when you code, you need to think that you are not the only one working on it, even if you are working alone as a single dev army, that will help a lot.

So let me give you some tools to write better code, first let's open your mind a little.

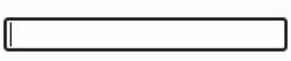
Atomic design, Front End point of view

Separate your code in the minimum piece of logic as possible, the smaller the code the easier to test. This also brings more benefits, like reusage of the code, better maintenance and even better performance; as the code gets smaller and better organized and depending on the language/framework we use, we could end in less processing cycles.

Maintenance will be greatly improved, as we are coding small pieces of work, each one with the loosest coupling and highest cohesion as possible, we can track and modify the code with the minimum number of problems.

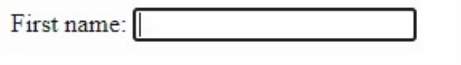
Let me show you how to think atomically, and how you can reach a complete app starting from a small input.

First we have our input:



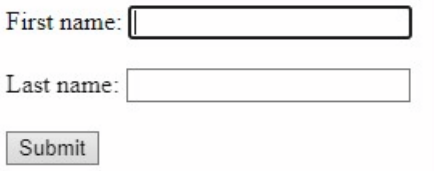
simple stuff, now that is what we call an Atom, the minimum piece of logic as possible. If you code it atomically you can reuse this input everywhere in your app and, later on, if you need to modify its behavior or its looks you just modify one little atom with the result of having an impact on the whole application.

Now, let's say you add a label to that input:

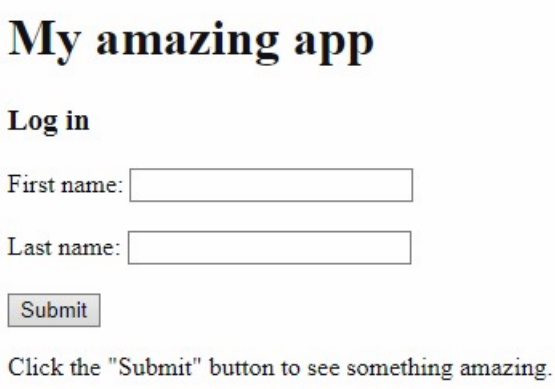


Congrats ! Now you have what it's called a Molecule, the mix between atoms, in this case a label and an input. We can continue going forward and reducing granularity.

We can use the input with the label inside a Form creating an Organism, the mix between molecules:



If we mix Organisms, we will get a **Template**:



And a collection of templates creates our Page, and then using the same logic, our App.

Using this way of thinking will result in your code being really maintainable, easy to browse to track errors and more than anything...easy to test !

If you write anything other than an Atom, it would be really difficult to test anything, as the logic would be of high coupling and therefore impossible to separate enough so that you can check specific cases.

One example would be testing a high coupled code, to validate just a simple thing one would need to start including one piece of code...and then another...and another, and after you finish you will see that you included almost the whole code because there were just too many dependencies from one place into another.

And that's the key to include a mvp (most valuable player) in all of this.

Functional Programming

So functional programming it's a paradigm that specifies ways of coding in a way that we divide our logic into declarative, with no side effects methods. Again...think atomically.

When we start learning how to code we normally do it in an Imperative way, where the priority is the objective and not the way we reach it. Even if it's faster than functional programming, which it is, it can bring a lot of headaches apart from leaving aside all the benefits from the other one.

Let's write a comparison in Javascript.

Imperative way of searching an element inside an array:

```
var exampleArray = [
    { name: 'Alan', age: 26 },
    { name: 'Axel', age: 23 },
];

function searchObject(name) {
    var foundObject = null;

    var index = 0;

    while (!foundObject && exampleArray.length > index) {
        if (exampleArray[index].name === name) {
            foundObject = exampleArray[index];
        }

        index += 1;
    }

    return foundObject;
}

console.log(searchObject('Alan'));

// { name: 'Alan', age: 26 }
```

And now the functional way of reaching the same objective:

```
const result = exampleArrayMap.find(element => element.name === name);

console.log(result);
```

Is not only shorter, it's also scalable. The map method we are applying into the array is a declarative one from ECMAScript, that means that every single time we run the method using the same parameters, we will always get the same result. We also won't modify anything outside the method, that's what's called side effects, the method returns a new array with the elements that comply with the condition.

So if we create methods representing the minimal units of logic as possible, we can reuse working and tested code across the app and maintain it if needed. Again...think atomically.

Now that we know the way of thinking to create high quality and easy maintenance code, let's go into what an User Story is.

🕶 User Story and TDD

What a title right ? Everyone knows about user stories, how to define them, what we need to do with them, but no one follows the same way of writing one or even its structure.

A user story is the explanation of a feature from an user point of view. It normally looks something like this:

What? As an user (who), I want to have the possibility of...(what) to...(why)

How? (use cases) 1- step 1 2- step 2 3- step 3 ...

So as you can see, we define who...the user, what he wants to do...the functionality, why we want this functionality...which while writing it we can even discover that it doesn't even make sense creating it because the objective is not clear, and how we will create it...the use cases.

The use cases represent the number of requirements that we need to fulfill to clearly say that a user story is done, they normally tell the story of the happy path to follow. There are also places where the entities related to the user story and the corner cases (sad path) are also described inside them and I believe that's a really good practice, but the same as when writing high quality code...we need to identify the boundaries of our context to see how can we write the content as specific as possible without transforming our task into a really difficult to follow and time consuming document.

Now TDD, Test Driven Development, it's a process where we define our tests before even coding a single line, so...how do we test something that's not even created right ? Well, that's the magic of it, you can grab your use cases and define what you need to reach each one of them, create tests around them, make them fail, and then fix them to pass..simple as that.

The main idea behind TDD is to think about:

- What do you want to do?.
- What are the core requirements?
- Write tests that will fail around them.

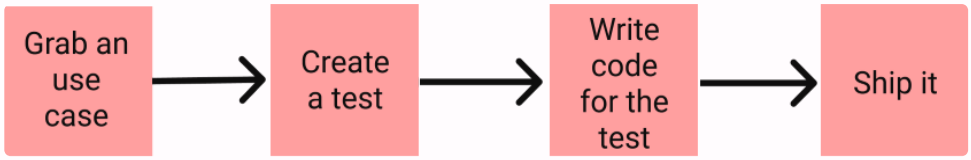
- Create your code knowing what you want to do.
- Make the test pass.

If you think testing is time consuming, well it is, but because you may have previously done it in the classic bad way of first coding everything and then trying to test your code. Remember what we were talking about code quality, good practices, etc ? Well, those are the main elements that will help you test your code and if you are not implementing them correctly we will end in an impossible to separate and test functionality.

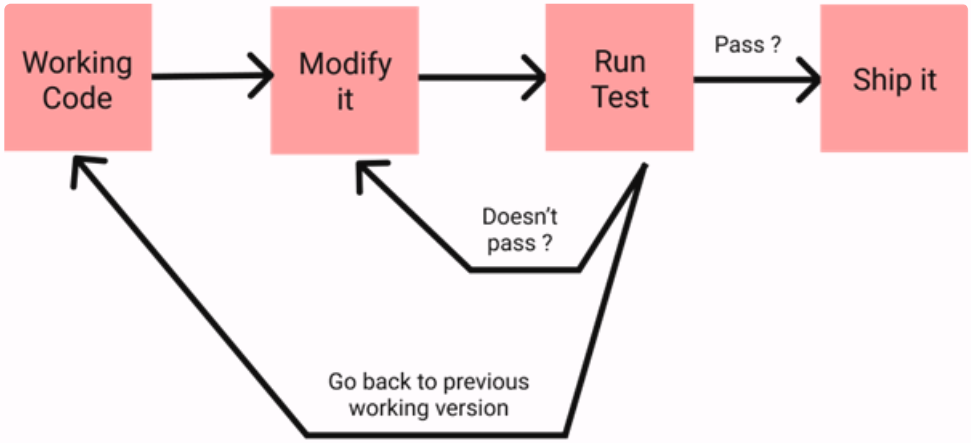
That's why coding knowing what you want to test, using functional programming and an atomic way of thinking can be so beneficial, because you will be creating logic, pinpointing the requirements and in the end... increasing the coding speed.

So here it is, testing also helps increase the coding speed, as you write more manageable code, it's easier to modify a requirement (use case) of your functionality as you have it identified by a test that will tell you if your refactor went correctly. It also reduces the possibility of bugs, so less time fixing problems later on.

TDD flow:



Here's a TDD flow on how to improve code quality without breaking anything:



Communication First and Foremost

We are reaching a new era! Remote work is coming strong and as the commodities increase also problems communicating between distributed teams.

Companies are not grounded to a certain location

Companies are not grounded to their office's location as they now play between the whole world's rules, so we need to change the mindset to understand this new paradigm. A great example is giving a job offer to a candidate, if we consider a salary limited to the candidate's location we run the risk of it being declined as he may have received offers from all over the globe, more tempting and better paid.

This concept may induce problems at any organizational level, people will compare themselves and what they do to professionals from all over the world and may think that they are not being offered the same level of benefits or that they just are not being paid enough. So how do you deal with this problem? making them feel part of something awesome, helping them overcome blockers to personal growth and most important of them all, keeping them learning new stuff.

Not everything that shines is gold goes the phrase, and we can apply the same concept between a company and its employees. People do not always search for money! knowledge is one of the greatest values one can provide as I always preach the following: “Knowledge first, money second; The more you know, the more someone is willing to pay you for it“

Coordinating across time zones

Being a distributed first company is not an easy task, managing work synergy across individuals that are not even in the same time zone can result in a bigger challenge than anticipated.

I recommend learning how to work asynchronously, it is just putting the puzzle pieces together, but the challenge is to find which are those pieces. Along with my professional experience, I detected that the most important and also difficult piece is generating a balanced amount of context among the team.

The way I have been doing it over the years is by understanding that communication is key and secondary to none, your team needs to be in the same boat or it will not work.

Second, you need to know your sources of truth, a place where you can check to clear doubts and search for context because it's always up to date. From my experience I found that there are two different kinds:

- One represents the reality of each business logic, a great example is the usage of Notion or Confluence as sources of truth, where we detail what we expect, why we are doing what we are doing, requirements, corner cases, etc.
- There's also the need for a second type related to workloads and what the team has agreed to do, and is one you already know, the created tickets that the team will complete over the passage of time.

We need two different types because while the first one gives us all the context we may need, we also have the necessity to put a stop at some point and decide when to start the implementation process. This last one details what the team has agreed on doing with the correct amount of context so they can provide enough value and don't be blocked in the process, and we need it so we can continue working on improving and evolving.

Let's think about what happens when having doubts about a certain task, a lot of people would create a comment inside the ticket, leave a message inviting communication over a chat channel, etc. This usually ends in a lot of meaningful chatter, confusion and more doubts, as we can't fully understand the intention of the text. How to fix it? Just sort the order of the elements in a different way:

- Ask for a call, talk with each other and adapt in a way that needs are fulfilled on both sides.
- Document the results over the related ticket and leave a history that can be tracked for future similar situations.
- It's ok to use chat channels for simple subjects but move into a meeting as soon as you see that the conversation is going nowhere.
- Meetings are not meant to add context or new information, please update the source of truth and share it as a first choice.
- Meetings are not meant to add context or new information, please update the source of truth and share it as a first choice.
- Include the related source of truth link inside the tickets, but also try to add all the needed information from it directly into the ticket to leave a definition of what has been agreed at the moment of creation.

There are some other easy ways we improve this mindset

We can start by over-communicating decisions across all geographies, this will result in people understanding what's going on and the whys.

Minimizing the friction in setting up a work environment, having documentation and some sort of guidelines will improve new additions to the team processes of getting to know the way they are expected to work and getting up to the required level to start working.

Clearly define the definition of done, this one relates to the need of having a workload source of truth, what are the acceptance criteria we need to complete to move a task as "done". Also, remember what I said before, a feature needs to have a start and an end of its own in a way that can provide value while being independent.

Using some of the concepts already provided in the previous chapters, for example doing pair programming or code reviews helps distribute knowledge between offices, helps generate a structure between global teams and minimizes the amount of collaboration required.

Creating Rapport

Reaching an accepted level of affinity between distributed teams can prove to be challenging but there are some things we can do to increase our possibilities:

- Communicate even minute details until both offices find a healthy groove.
- Communicate decisions.
- Everyone needs to understand the decision and why it was made.
- Don't use emails as they are an easy way of losing information.
- Use a content management system, for example, a wiki.
- Create channels for individuals and teams to communicate and see updates, another great idea would be to create channels for a certain future and the involved people.
- Spend time creating a simple but effective "Getting Started" guide.