Pragmatic Penguin Patterns

Pragmatic Penguin Patterns: become a better software developer

version: 2.1.0

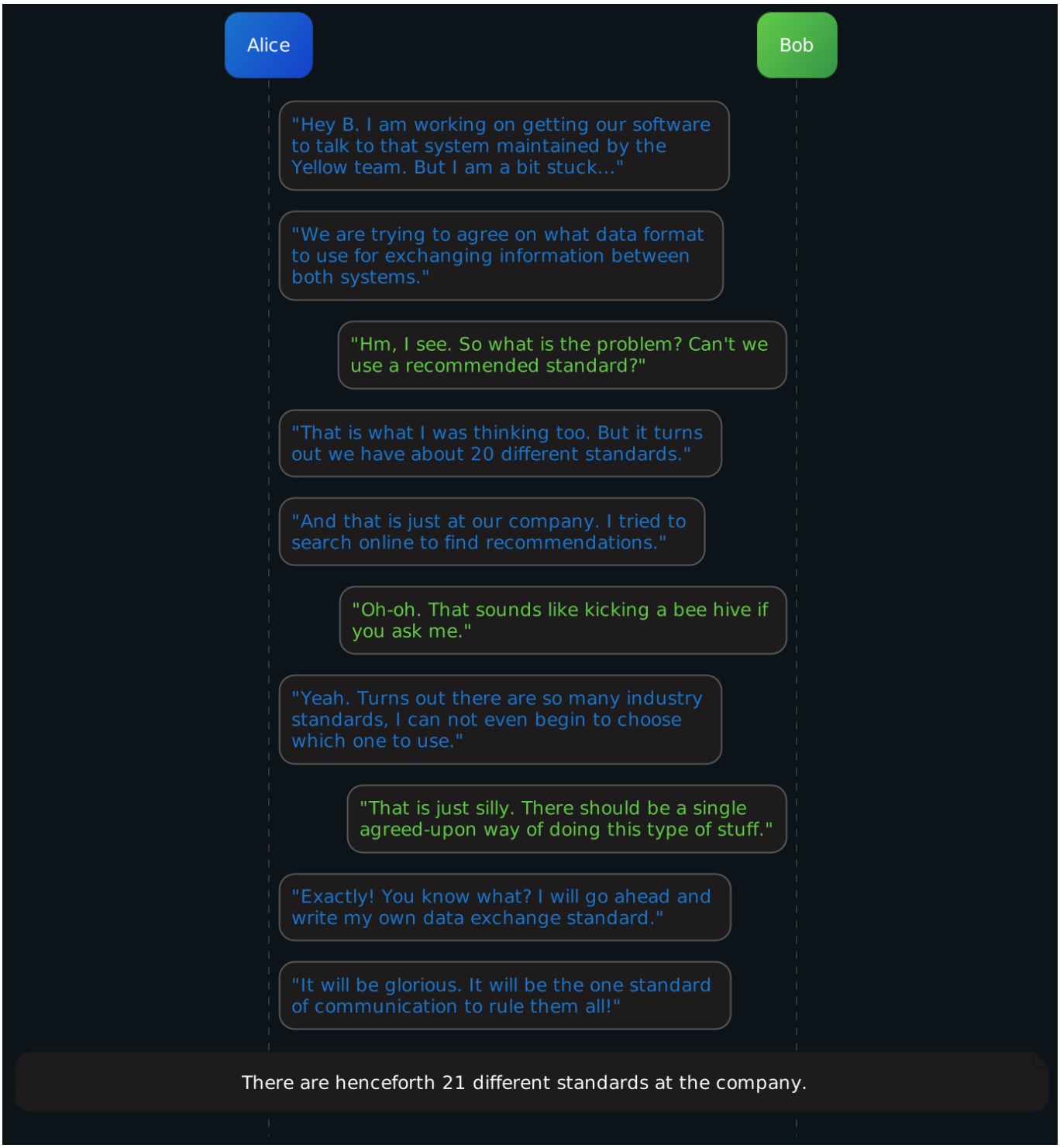
Hello world!

Are you interested in learning how to be better at the things you do on a daily basis? Are you working in a technical, or mostly technical role? Or are you just interested in picking the brains of those who are?

If so, keep reading as this piece of writing could be of value for you.

Over the years, I have consumed a lot of books, articles, and videos with the goal of improving my skills as a technical professional. Some lessons I have learned during this time have proven to be great assets time and time again. Others have been mostly liabilities to my work. I have had the pleasure of working with some very capable, and motivated developers. But I did notice that what our industry lacks most is common ground. Other professions are more mature than ours. Which is only natural, as they have been around for ages. Their By comparison to crafts such as woodworking, medicine, or even politics, software development is a fairly young profession.

At the time of writing (late 2022), the technology industry has been steadily growing in size and importance year-on-year. Us nerds are now at the point where we can safely say that we control most of the world. While this does sound like a dream come true for our 10-year-old selves who fantasized about living in cyber-space and being all-powerful wizards, the reality is more grim. Right now around 20% of people working as software developers have less than two years of experience[^0]. As our industry doubles in size every few years, this number is only expected to rise as time passes and technology becomes more important.



This rapid growth is one of our biggest strengths. On the flip side, it is also our biggest challenge as an industry. We seem to be unable to come up with a good way to express our ideas, and even less able to communicate these ideas and our experience to the next generation of software professional. The internet is filled with information on how to write code, how to make things works, and a plethora of "quality of life hacks" that are supposed to make our lives as developer a lot easier. But when going online, you will also see that if you put two developers in the same room, they will have three opinions on what the right way to do things is.

Purpose

This work is a learning and development knowledge base, aimed to share knowledge with other technical professionals. The docs section of this repository acts as somewhat of a “personal wiki”. It contains

knowledge and information that has been collected from various sources, and is extended with some personal interpretations and experiences of the author.

As I stated earlier, there are a lot of great books on the topics I will discuss in this course. Most of them do a much better job of explaining the concepts than I can ever hope to do. This work is a high-level introduction to concepts, and practices, stating why there are important. I aim to give enough information to use them practically. At the end of each lesson, I will include a list of resources for further learning. I invite you to dig deeper into the topics you find interesting.

As we live our lives, we tend to learn a great deal about a wide variety of topics. From time to time we are stumped by how elegant, or easy, someone solves a certain problem. You are left wondering why you have not been tackling similar problems in the same way. It could have saved you vast amounts of frustration, if only you had known earlier.

This catalog aims to bundle little nuggets of enlightenment these aha-moments tend to deliver. We hope that someone, somewhere learns something that makes their lives a bit more enjoyable.

> [!NOTE] As this knowledge base will likely remain a work in progress for some while, not all content will be refined. It is advisable to check out the Changelog in order to see what has changed recently.

Intended Audience

This knowledge base is intended for people that are interested in improving theis current way of working (or living) by learning about alternative approaches. You will likely already be familiar with some ideas in this knowledge base, in which case: feel free to move on. If you think we are blabbering and giving bad advice, feel free to leave a comment or suggestion on the github page.

Structure

This open knowledge collection is ordered by categories. Each category contains some of the following resources:

- **A collection of Patterns:** Little snippets of information to inspire you. These can be used as general pieces of advice that may or may not be applicable to your current situation. To maximize their usefulness, these patterns (or motifs) follow a similar structure.
- **Articles and short-form information:** These sections contain a list of articles that were written by either myself or others and added with their permission. These articles can be opinionated pieces, additional context, or specific case-studies.
- **Resources and Reviews :** Additional information related to a topic, that do not easily fit the other sections. These might include book recommendations, tutorials, tool recommendations, etc.

A set of concepts and common abbreviations can be found in the glossary.

If you are interested in diving deeper into some subjects mentioned in this work, be sure to take a look at the Reading list section included in the appendices. There, you will find an overview of great books, articles, and audiovisual recourses that are worth checking out.

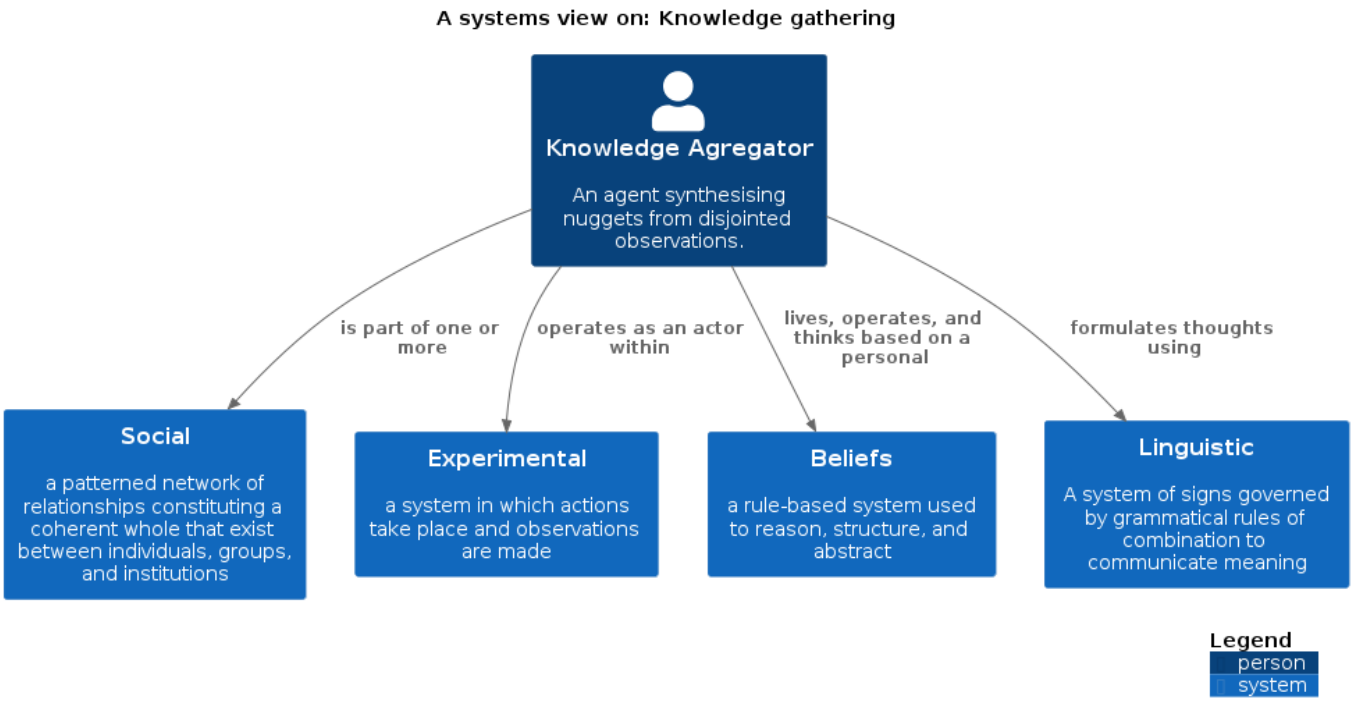
> [!NOTE] The information in this knowledge base is to be seen as a series of techniques that have worked well for people in the past. However, they are not *'recipes'*. Meaning that you are encouraged to take the ideas and adapt them to fit your personal context.

A pattern-based approach

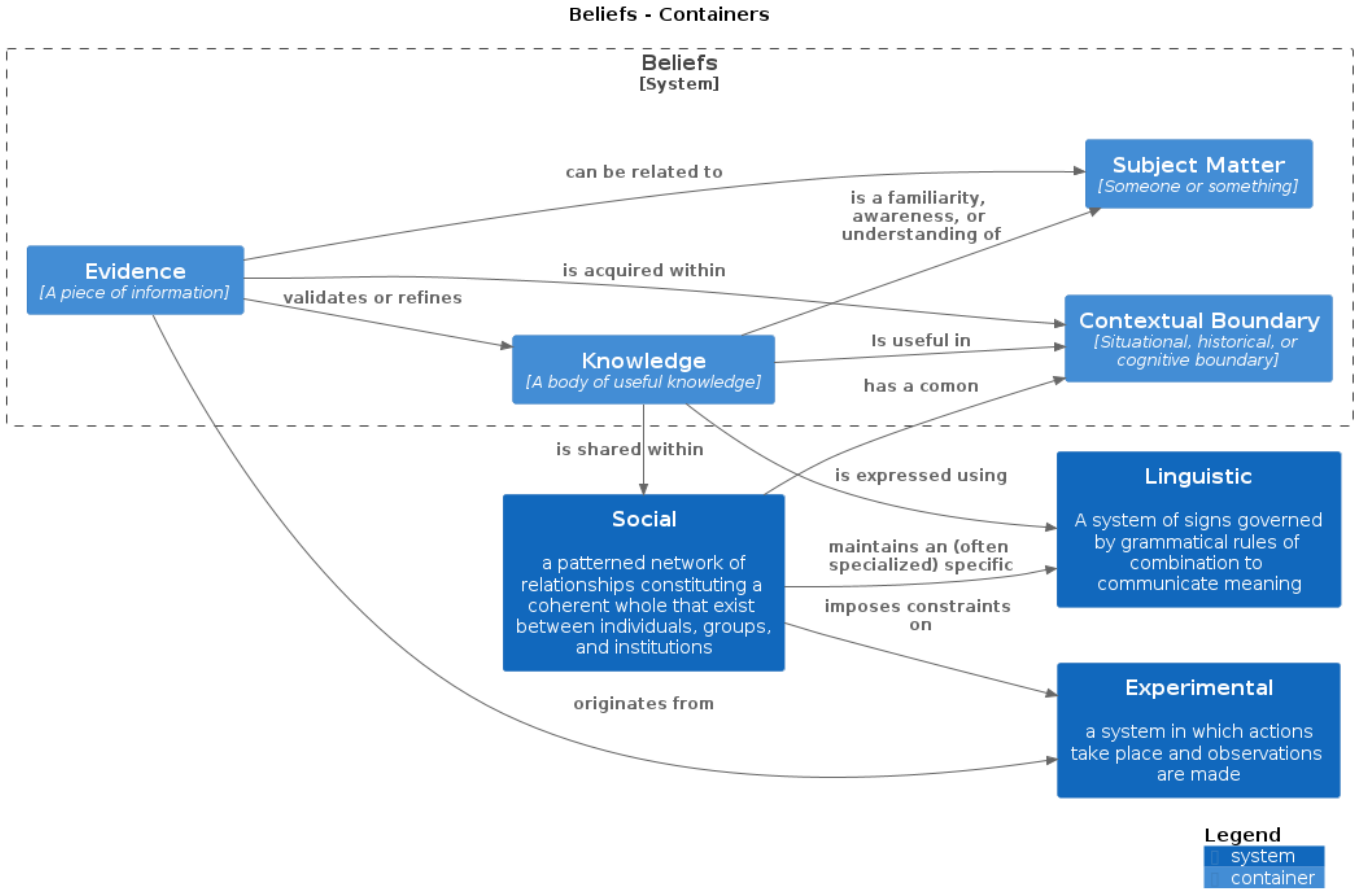
Exchanging knowledge is a challenging activity. The aim is to communicate thoughts and ideas in a way that makes them resonate with the receiving party. To communicate effectively, groups of people tend to resort to using jargon. The term `jargon` means: *"The specialized language of a trade, profession, or similar group, especially when viewed as difficult to understand by outsiders."*. Its goal is to make the exchange of information more efficient by giving specific names to things that are relevant to the in-group. It is said that the Inuit have over thirty words to differentiate between different types of snow. Other professions, such as software developers, strongly rely on metaphors to refer to technical concepts.

When looking for a structured way to represent ideas, experiences, or cookbooks (i.e. knowledge), it helps to first take a look at how we see *"knowledge"* itself. Us software developers tend to resort to modeling things when we want to understand them a bit better. By visualizing our ideas, and how they relate to each other, we create a "map" of how we perceive a certain set of concepts.

Knowledge representation



The image above is an ideation map, showing us how a person gathers information. You will see how our "knowledge aggregator" interacts with a few relevant *"systems"*. From this simplified visualization, we see that how one perceives information is dependent of oneself, and the context one lives in.

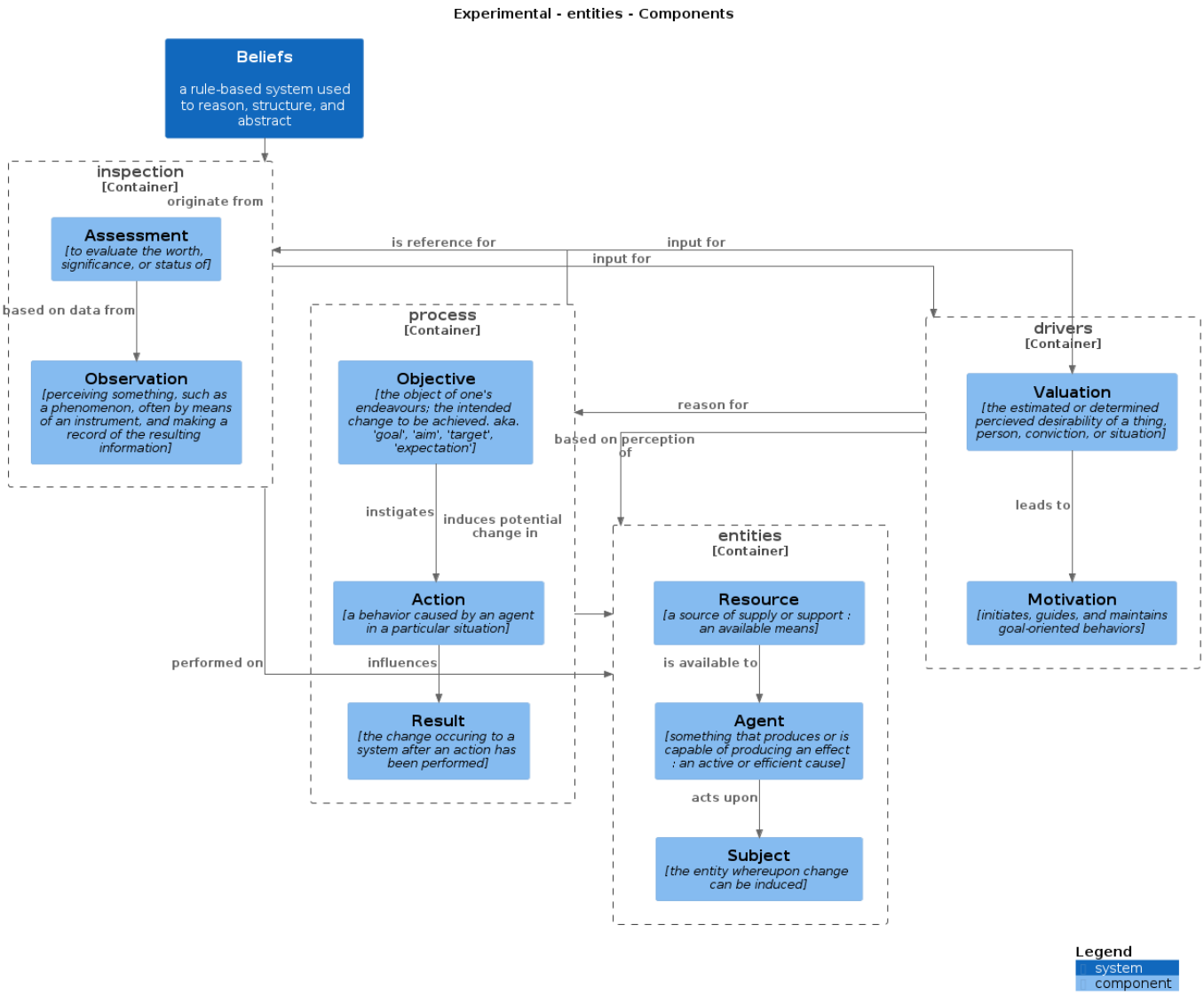


When you talk to people, you rely on a shared understanding of the language and terminology you use. The words you use will vary depending on whom you are speaking to, what you are discussing, and even how you are feeling at that particular time. Even more interestingly, how we talk about things tends to influence how we think about them. This means that if you spend most of your time complaining about something, you will perceive it as being more of a pain than it actually was before you started communicating negatively about it.

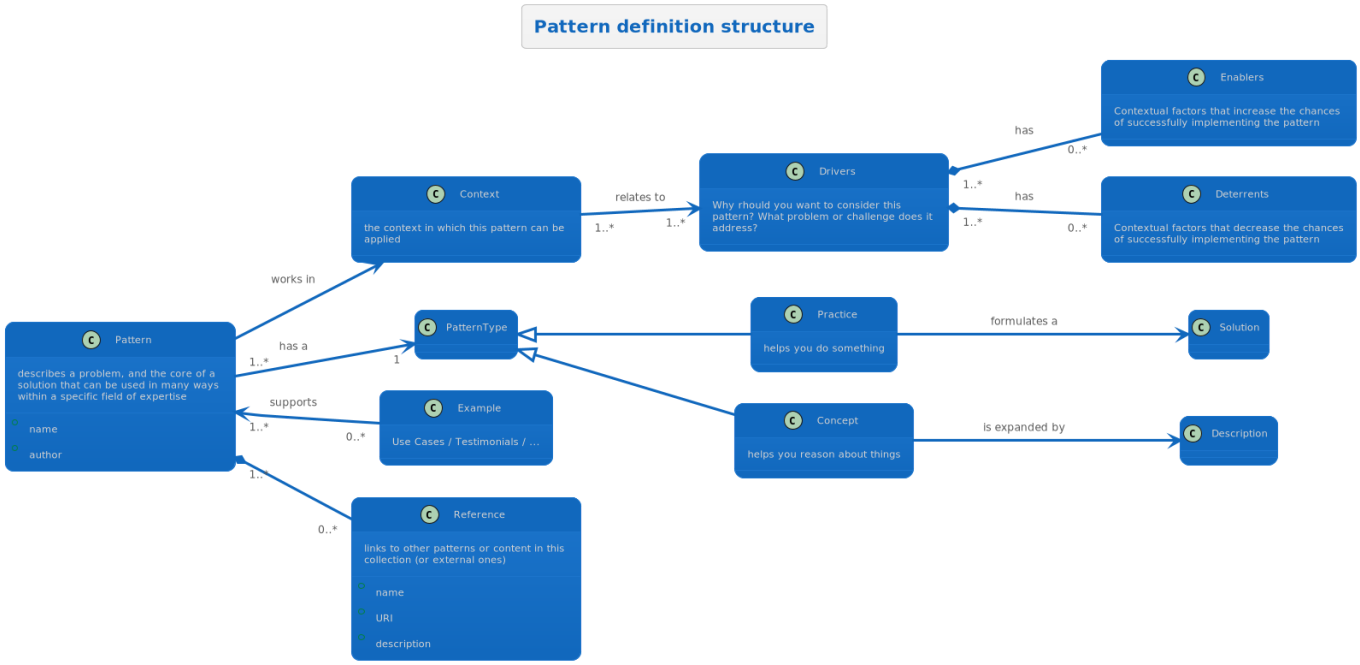
A pattern language is **a formal way to represent wisdom that improves ones ability to operate in a certain field of expertise**. The knowledge represented in a pattern language is usually stripped down to its bare essentials, in hopes of making it easier to apply them in a variety of situations.

Much like a dictionary, each entry (*"Pattern"*) in these knowledge bases is indexed with a `name` and contains information on `what challenge it addresses` as well as a `description of the context in which it works well`.

Sharing knowledge with other practitioners in a formalized manner is an activity that is generally used in a variety of fields. Chess players share certain board positions and common tactics by manner of "motifs"[^1] . In popular internet culture, we see the same as plenty communication happens using memes and tropes[^2] . Recent marketing campaigns, and elections in various countries have seen an uptick in using funny pictures and videos on the web to influence peoples decision-making.



We want to share information about both things you can do, and about ideas and models that can change the way you perceive a situation. To achieve this, the patterns in this knowledge base are split up into: `Concepts` and `Practices`. To keep the content in this work consistent, the patterns follow a similar structure. As we now know, **context matters**. This is why each pattern is prefaced with a short description of when it can be useful to consider using it.



Some patterns will contain references to other resources. They can also contain supporting examples, testimonials, tutorials, etc. These examples can be included inside the pattern description, but they might as well exist as a separate resource in a different section of this publication.

Happy reading, we hope you learn something helpful. If you did, please pass on what you have learned to someone you know that might find your newly gained knowledge helpful.

---

[^0] Statistics retrieved from [https://www.zippia.com/software-engineer-jobs/demographics/] (https://www.zippia. com/software-engineer-jobs/demographics/), Evolution of software engineer demographics in the USA in 2019.

[^1] Unfortunately knowing this does not instantly make you a great chess player, as my elo on online platforms so adequately reminds me.

[^2] So wide-spread even, that academics have started investigating what makes some memes successful while others are doomed to be left in the ditches of the web.

1_Learning

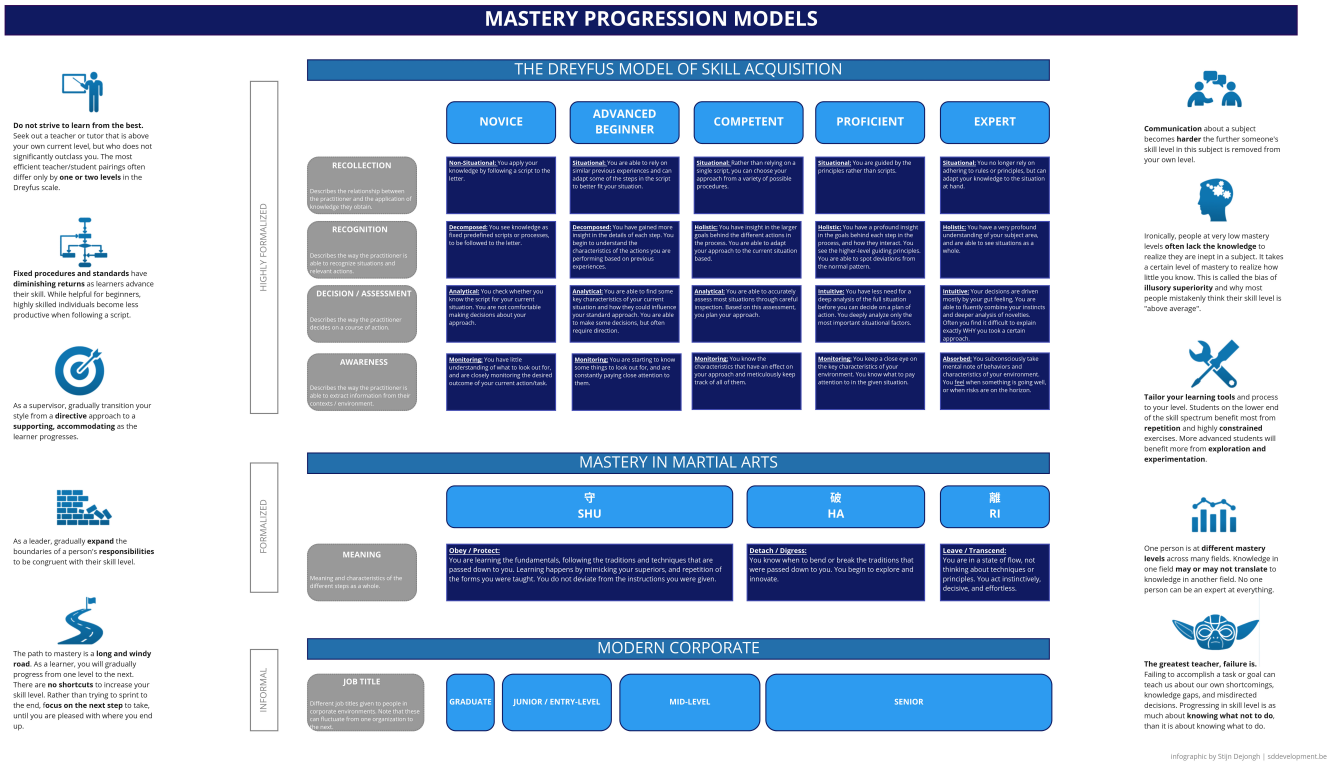
`/1_Learning`

> [!WARNING] **This is not a bingo chart.** Unlike when playing Pokémon, you will not get a special prize for getting all the patterns in this publication into your work or life. You are encouraged to be eclectic, and only use the practices that resonate with you. Feel free to change them as you see fit, after all: Who are we to tell you what to do?

Why learn about learning?

Over the ages, a lot of research has been done on just how people are able to learn and acquire new skills. As humans, we often feel a need to know how good we are at something, and how to become better. In the following sections, we will go over a few of the more useful conceptual models on the stages of mastery. In

the modern knowledge-based economy, being apt at learning new skills and concepts is a significant competitive advantage.



Meta-Learning

Meta learning is originally described by Donald B. Maudsley (1979) as "the process by which learners become aware of and increasingly in control of habits of perception, inquiry, learning, and growth that they have internalized". Maudsley sets the conceptual basis of his theory as synthesized under headings of assumptions, structures, change process, and facilitation. Five principles were enunciated to facilitate meta learning. Learners must:

- have a theory,however primitive
- work in a safe supportive social and physical environment
- discover their rules and assumptions
- reconnect with reality-information from the environment
- reorganize themselves by changing their rules/assumptions.

(source: wikipedia: Meta-learning)

Concepts

`/1_Learning/Concepts`

Dreyfus model of skill acquisition

> Every journey begins with one step in the right direction. In order to know the direction to take, it is important to know where you are at right now.
>
> ~ paraphrasing Sun Tzu

Description of Pattern

The Dreyfus model of skill acquisition is a very formalized model, that defines 5 distinct steps of mastery. As a learner progresses from left to right, their way of using the skill in question changes. Even more interesting is that they start thinking in different ways.

	Novice	**Advanced Beginner**	**Competence**	**Proficient**	**Expert**
Recollection *Describes the relationship between the practitioner and the application of knowledge they obtain.*	Non-Situational *The learner applies their knowledge by following a script to the letter.*	Situational *The learner is able to rely on similar previous experiences and can adapt some of the steps in the script to better fit their situation.*	Situational *Rather than relying on a single script, the learner can choose their approach from a variety of possible procedures.*	Situational *The learner is guided by the principles rather than scripts.*	Situational *The learner no longer relies on adhering to rules or principles, but can adapt their knowledge to the situation at hand.*
Recognition *Describes the way the practitioner is able to recognize situations and relevant actions.*	Decomposed *The learner sees knowledge as fixed predefined scripts or processes, to be followed to the letter.*	Decomposed *The learner has gained more insight in the details of each step. They begin to understand the characteristics of the actions they are performing based on previous experiences.*	Holistic *The learner has insight in the larger goals behind the different actions in the process. They are able to adapt their approach to the current situation.*	Holistic *The learner has a profound insight in the goals behind each step in the process, and how they interact. They see the higher-level guiding principles. They are able to spot deviations from the normal pattern.*	Holistic *The learner has a very profound understanding of the subject area, and are able to see situations as a whole.*
Decision *Describes the way the practitioner decides on a course of action.*	Analytical *The learner checks whether they know the script for the current situation. They are not comfortable*	Analytical *The learner is able to find some key characteristics of their current situation and how these could influence*	Analytical *The learner is able to accurately assess most situations through careful inspection. Based on this*	Intuitive *The learner has less need for a deep analysis of the full situation before they can decide on a plan of action.*	Intuitive *The learner's decisions are driven mostly by their gut feeling. They are able to fluently combine their*

	making decisions about their approach.	*their standard approach. They are able to make some decisions, but often require direction.*	*assessment, they plan their approach.*	*They deeply analyze only the most important situational factors.*	*instincts and a deeper analysis of novelties. Often they find it difficult to explain exactly WHY they took a certain decision.*
Awareness *Describes the way the practitioner is able to extract information from their context / environment.*	Monitoring *The learner has little understanding of what to look out for, and is closely monitoring the desired outcome of their current action/task.*	Monitoring *The learner is starting to know some things to look out for, and is constantly paying close attention to them.*	Monitoring *The learner knows the characteristics that have an effect on their approach and meticulously keeps track of all of them.*	Monitoring *The learner keeps a close eye on the key characteristics of their environment. They know what to pay attention to in the given situation.*	Absorbed *The learner subconsciously takes mental note of behaviors and characteristics of their environment. They **feel** when something is going well, or when risks are on the horizon.*

download infographic

Usage

Tailoring coaching approach to the learners skill level

In general, the more experienced one becomes, the more they start internalizing a concept of "the bigger picture". Inexperienced practitioners of a skill are pleased when someone provides them with clear-cut instructions, especially if they've heard applying these instructions will lead to certain success. Adapt your coaching and teachings styles to the needs of the learner.

Compatible with

- Most incremental improvement techniques
- Self-reflection patterns
- Personal knowledge management

References and further reading

author	url	title	publisher
Dreyfus, Stuart E;	http://www.dtic.mil	A Five-Stage Model of	Storming Media

Dreyfus, Hubert L		the Mental Activities Involved in Directed Skill Acquisition	
Hunt, A	pragprog.com	Pragmatic Thinking and Learning: Refactor Your 'wetware'	Pragmatic Bookshelf
Hunt, A; Thomas, D	pragprog.com	The Pragmatic Programmer, your journey to mastery	Addison Wesley/Pragmatic Bookshelf
Hoover, D; Oshineye, A	oreilly.com/library	Apprenticeship Patterns: Guidance for the Aspiring Software Craftsman	O'Reilly Media, Inc

Shu Ha Ri

Shu Ha Ri is a Japanese martial art concept that is used to describe the stages of learning on the path to mastery.

Description of pattern

> When we learn or train in something, we pass through the stages of shu, ha, and ri. These stages are explained as follows. In shu, we repeat the forms and discipline ourselves so that our bodies absorb the forms that our forebears created. We remain faithful to these forms with no deviation. Next, in the stage of ha, once we have disciplined ourselves to acquire the forms and movements, we make innovations. In this process the forms may be broken and discarded. Finally, in ri, we completely depart from the forms, open the door to creative technique, and arrive in a place where we act in accordance with what our heart/mind desires, unhindered while not overstepping laws.
> ~ sensei Seishiro Endo 2005

Shu 守: Obey / Protect

You are learning the fundamentals, following the traditions and techniques that are passed down to you. Learning happens by mimicking your superiors, and repetition of the forms you were taught. You do not deviate from the instructions you were given

Ha 破: Detach / Digress

You know when to bend or break the traditions that were passed down to you. You begin to explore and innovate.

Ri 離: Leave / Transcend

You are in a state of flow, not thinking about techniques or principles. You act instinctively, decisive, and effortless.

Related Patterns and Resources

Item	Author	Description
Kokoro extends Shu-Ha-Ri with “Heart”	by Alistair Cockburn	:fas fa-ninja: :fas fa-trophy:

2_Productivity

`/2_Productivity`

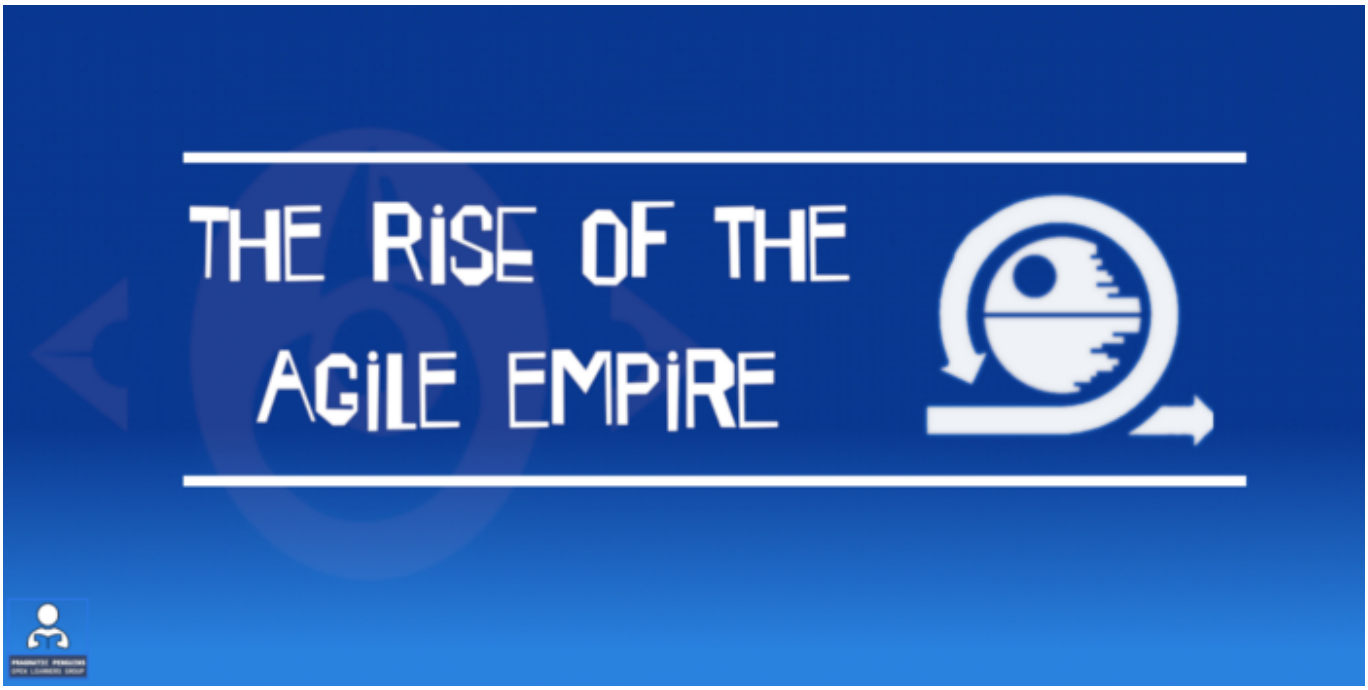
> [!WARNING] **This is not a bingo chart.** Unlike when playing Pokémon, you will not get a special prize for getting all the patterns in this publication into your work or life. You are encouraged to be eclectic, and only use the practices that resonate with you. Feel free to change them as you see fit, after all: Who are we to tell you what to do?

Articles

`/2_Productivity/Articles`

Why should you be concerned about the rise of the Agile Empire?

> A story about the dangers of blindly adopting “best practices”



Introduction

Not so long ago, in development teams all over the world, methodologies of ‘ye olden days’ are popping up. Project managers and senior developers are rediscovering the written wisdom of the giants in our fields. Past ideas have been given a second life. Alongside, newer derivative systems are emerging. Clever marketeers have found a way of selling their variations of agile processes as a silver bullet solution for struggling teams. Within this turmoil, a brave alliance of underground freedom fighters is challenging the tyranny and oppression of the awesome AGILE EMPIRE.

A new hope: agile software development

In the world of software development, there is an undercurrent of people that believe that *"if we can just find the right system, everything will go perfect"*. If anyone ever manages to come up with the perfect cut-and-paste methodology that can be applied to any failing software project, and instantly transform it into a high-performing team, they will make millions if the marketing is adequate.

For the last couple of years, methodologies of *'ye olden days'* are popping up left, right, and center. Project managers and senior developers are rediscovering the written wisdom of giants in our field. Past ideas such as 'pair programming', 'XP', 'Scrum', and 'Kanban' have been given a second breath of life. Alongside them, newer derivative systems such as 'DevOps' and 'SAFe' have emerged.

The downside of this resurgence is the incremental pollution of the original values of agile software development due to claims of one-size-fits-all solutions. Clever marketeers have found a way to sell their variations of agile processes as a silver bullet solution for struggling teams. In this article we will take a look at the original values, and mix them with my personal observations and opinions.

For those of you that are unaware of the origins of Agile Software Development, I'll start with a very brief look at the history of agile software development.

The story of agile software development

Around the start of the new millennium, the software industry was being frowned upon. First there was the dreaded 'millennium bug', that exposed hundreds of thousands of shortcuts taken by developers over the years. The result was that a lot of companies and governments were not sure their software would be able to cope with the new date that ended in double zeroes. Representing dates in software code has always been a challenge. Just before New Year's Eve 2000, auditing companies around the world started reporting that the majority of software solutions were likely going to crash and burn.

The reason for this was that a bunch of developers had used the last two digits of the Gregorian calendar's years in date representations and calculations. This meant that affected software would assume the current year was actually '1900', instead of '2000'. This led to the world as a whole looking at our cozy IT bubble with a magnifying glass. Stories of projects gone bad, expensive bugs, and movies demonstrating how the world would basically end if we did not get our stuff together, came to the foreground of news reporting.

The increasing scrutiny, and the general feeling of discontentment with their current processes, led to a few bright developers creating their own systems. These systems were a structured bundling of the lessons these development gurus had gathered over many years of their professional careers. As with all thing IT, debates began about which system was the best. Developers all over the world started looking for the *"one system to rule them all"*.

Responding to this, that bright few who created the most popular systems, decided to get together at a ski resort to discuss their shared beliefs. This free-form discussion ultimately resulted in The Manifesto for Agile Software Development. One of them made a simple website with nothing more than their four core beliefs, and a set of complimentary principles that demystify the four core beliefs. The development world jumped on this wisdom like a pack of starved dogs on a steak. The manifesto and principles started to become revered as the Holy Bible of software development. The values and principles were regarded as commandments, more so than a set of good advices.

Our creed is defeated by its own success!

At first, it were mostly software developers that bought into the manifesto. They held conferences, and tried to deliver software in a better and more relaxed way. The few, who were allowed the freedom to work as they pleased, reported good results. Fast forward a decade or two, and we see an abundance of agile consulting firms. Each offering training in their "optimized method of agile". The agile movement started feeling more and more like an industry of cultist preachers, like you would see on American TV shows, than as a group of developers trying to deliver good software. Many of these consulting firms market their agile methodologies as the end-all-be-all of project management. As such, a lot of project leads and managers got very interested in these methods that claim the team will be able to cope with any change of requirements, will always have good estimates, and always deliver on time, no matter what happens or who leaves the team. I must admit it sounds like a wet dream to professionals in a leadership position.

Nowadays, other industries are buying into the agile hype. **And they are buying in hard!** You see companies rolling out their own version of SCRUM with their company branding and flavor applied to it. Some use these processes as their core selling point to customers: *"We are different, we are Agile! So we will always deliver on time!"*. People that look for a job in the software industry are routinely asked if they have experience with Agile/SCRUM/Kanban. Saying you have experience with these methods has become a big competitive advantage for both companies and individuals.

The downside of all of this success is that the big majority of Agile practitioners have never heard of the original manifesto, or the values it champions. They have no clue that the original idea was to provide a framework for a personal creed and a way of working specifically aimed at developers. Instead, we ended up with the rigorous application of as many SCRUM "ceremonies" we can cram into our days, a plethora of competing certification systems, and a bunch of gurus claiming their flavor of SCRUM is the only right way to do software development.

Recently, my developer heroes, the pragmatic programmers (Dave Thomas and Andy Hunt), have started proclaiming they regret how popular the 'Agile' system they co-created has become. Dave gave a very insightful and interesting talk on the subject at the GOTO conference in 2015, basically pleading for people to stop sheepishly following the latest and greatest flavor of SCRUM, and to grab back to the original values of the Manifesto.

The manifesto strikes back

I fully agree with Dave when he says that the branding of Agile has taken a turn for the worse. Let's take a look at the values that were championed by the original authors of the Manifesto for agile software development, as published on agilemanifesto.org. The authors of the manifesto discussed for days to find the perfect wording, they refactored their initial text countless times to come to a manifesto that had "all the right words, in all the right places". This is the reason that their website contains the line *"this declaration may be freely copied in any form, but only in its entirety through this notice"*. Unfortunately, the last sentences and the header of the text are often omitted or regarded as of little value.

The original values

I will dissect the values from the original publication, and offer some personal annotations to them.

> We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:

The authors are addressing software professionals that are looking for guidance to write good software. With this paragraph, the authors are clearly demonstrating that the set of high-level values that follows originates from personal experience in the field. As we mentioned before, these are personal beliefs and values to be applied when the situation calls for it, not holy commandments.

Individuals and interactions over processes and tools

It's great to have tools and processes that help your team communicate in a structured and clear way. The idea of speaking the same language, and using the same formats, is of great value to teams far and wide. The downside of this way of working is what we in Belgium call the *"Over the wall"*-principle. It means that an individual in a team does not feel like part of a team, just does what he is asked, and then throws the task over to the next person (on the other side of the wall). Ticketing systems are notorious for enabling this behaviour. If you ever hear the words *"The ticket isn't formatted correctly, so I am not able to help you"*, chances are you are working in somewhat of an "over the wall"-process. Even teams that claim to *be doing agile* often fall into this trap. They tend to be overly reliant on their tooling (Jira, Kanban boards, excel sheets, ...) and lose focus of what these tools are there for: **communicating with each other**.

The authors of the manifesto say they believe that simply talking to each other, being open in your communication, and informally sharing ideas, make for a more efficient and relaxed way of sharing information. The back and forth dialogue that results from just speaking to each other often uncovers hidden, or unspecified knowledge. In more formal systems, information that is missing grinds the process to a halt: analyses have to be redone, the specs needs to be rewritten, etc. If our core process is to have more **casual and honest conversations** about what we are doing, such **costly mistakes can easily be avoided**.

Working software over comprehensive documentation

At the end of the day, we want to build something that works. In certain methodologies, schematic drawings, long functional descriptions, and presentations about software that does not exist yet, are seen as deliverables. This means that a large part of what the client expects and pays for is highly detailed documentation. The question rises if this really brings value to them. Would they be better off if you invested that time spent on documentation in writing code for a piece of the application they hired you to build? Sometimes development teams spend more time on keeping their documentation up to date than on actually writing code. That sounds like a team which has their priorities messed up, doesn't it?

Customer collaboration over contract negotiation

This core value is, in my opinion, a logical consequence of combining the first and second values and applying them to interactions with your client instead of interactions within your team. Some projects start with having all the requirements pinned down and checked off by the client. They then dive into their coding den and start churning out code and documents. Months (or years) down the line, the specifications have all been met, and the project can be presented to the client. It is often at this time the client realizes that they had other expectations of the software. This is followed by back and forth bickering, along the lines of: *"This is not what we paid you for!"*, *'Yes it is! Look, we have it in writing, and your signature is on the bottom of the page'*, and so on.

What matters most is that the customer is satisfied with the software you deliver. Whether that corresponds to what they originally asked of you or not, is not very relevant at the time of delivery. A satisfied customer will write a good review, or recommend you to their peers. A customer that is strong-armed into accepting and paying for something they did not want, will not be inclined to give you the same courtesy. The question here

is *"what hurts you the most?"*. Is it putting in extra effort to make sure the client is happy, or dealing with a bad reputation down the line?

> Responding to change over following a plan

Again, as in good code, the 'read down principle' is applied again. Again, this core value is a more specified version of the one before it. If you are collaborating with the customer, and showing them your progress on a regular basis, they will often times come up with alterations to the original specifications. *"Oh that's cool! You know what would make it even better? If we also added X and Y to this module, because it's very similar from a business perspective."* Responding to feedback like that, and making changes to your plan help you keep your customer pleased with your work. If you are willing to deviate from the "6 months plan" when the customer requests it, you will be building a solution that they will be happy with.

Other times, the customer changing their minds is not the only driver for change. Occasionally, your developers will discover that the framework they started using two weeks back is actually impeding their progress. The framework might even make it impossible to satisfy the next planned customer request. It's in times like these that you will end up delivering more value, by changing your course. Note that "responding to change" does not mean that the team will be able to deliver any random idea the client has. **Time and effort** are still a reality. If *"being able to deal with anything, and still deliver on time"* is why you started using an agile approach, you will be very disappointed or end up with mountains of software rot that grinds your project to a halt sooner or later. (let's not even think about developers running away to competitors because they are fed up with management's unrealistic demands.)

> That is, while there is value in the items on the right, we value the items on the left more.

In my opinion, **this is the most important sentence from the original text**. Unfortunately, it is the one most often forgotten.

The examples and notions I wrote down are simplifications of reality. Aiming to have more direct communication, having working software, *working with* instead of *fighting against* your customers, and being adaptable, are things to strive towards. This does not mean that having a contract in place, having some formalized processes, or even having a plan should be avoided altogether. Those things are also very valuable. prof. Rini van Solingen phrased this point very well: *"It is important to have a plan, because you need something to deviate from"*. You **need** a general outline, a map with a general direction to steer the ship towards. That does not mean you should sail into that huge iceberg, just because the map says that is the intended course.

Return of the agile developer

> In my personal opinion, we as an industry and us developers in particular, should aim to transition from "doing agile" to "being agile".

We started to notice that the combination of agile approaches and our modern *cut-and-paste* way of applying best practices can be a recipe for disaster. So, how can we do better? As I elaborately discussed above, there is no silver bullet approach to this. We should aim to be good at our jobs, and strive for agility in our daily jobs. I can tell you from personal experience that sometimes the most agile teams are the ones that officially follow a "waterfall process" but live the values of the manifesto on a daily basis. Your project structure will not stop you or your team from working in an efficient way. I doubt there is any boss or customer that will sit next to you, look at your computer screen, and tell you not to use TDD, or keep developer notes, etc.

In this section, I will take lessons from the concepts of pragmatism and craftsmanship to try and propose a different way of applying agile approaches.

An elegant Paradigm, for a more civilised time

A *"paradigm"* is a fancy word for "point of view". It is the belief system that you hold and use in order to make sense of what is happening around you. You can look at it as walking around in an unknown city, armed with a town map. If the map is not sufficiently accurate, you will find yourself ending up in the wrong place. You might even end up driving into a lake if the map is inaccurate, digital, and talks to you.

The word *'Pragmatic'* originally means "skilled in business". You can interpret this as thinking about the added benefit (return on investment) of an action before deciding to do it. A pragmatist will take pieces from various toolsets and methodologies, and apply them to the problem at hand only if it makes sense to use them. This means that even if a new software architecture is really hip, you would look at the issue you are trying to solve first and see if the new approach is worth doing.

> Take what you want, destroy the rest. ~from McCade's Bounty - William C. Dietz

A few years ago blockchain was all the rage. Many leading technologists wanted to use it on their projects, just to add a fancy buzzword to their sales pitch or resume. Back then, you would find articles all over the place claiming blockchain technology would solve any technological issue. In the end, a lot of time and effort was put in by developers, only to realise that their software had not become better by the inclusion of the new technology. Sometimes their software turned out to be slower, more expensive, and more confusing to their users. A pragmatist would likely have hesitated to jump on this hype train. He, or she obviously, would not do something just because everyone else is doing it. They would not use a technology if it added little value to the project they were working on.

Put the words *Pragmatic* and *Paradigm* together, and you know what I am getting at. Pragmatists look at processes, tools, frameworks, and life in general with an open mind and buy in to the things which help them progress, but change and amend the practices that deliver little value.

Mastery and Craftsmanship

To get to the bottom of the problem with blindly following advise from external experts, it's valuable to understand how one acquires skills. The Pragmatic Programmers (Dave and Andy) often use the metaphor of *workshops for craftsmen* in their publications to this end. People who view their vocation as a craft understand and appreciate that they will have to put in significant effort to excel at their jobs, they see themselves on the long road towards mastery.

We see this theme in the Dreyfus model of skill acquisition. Everyone starts of as a *'Novice'* and as their skills progress they go up in level. A few motivated individuals will eventually reach the level of *"Expert"* through hard work, commitment, and dedication. Note that while someone might be an expert in one area, he can very well be a novice in another area. The main difference between the lower and upper end of these levels is how they look at challenges, and how they apply their knowledge.

An expert will choose his approach mostly subconsciously, because it *"just felt like the right thing to do"*. Experts have the ability of putting their current task within a broader context and come to an adequate solution without the needed for overly analysing the situation. **Context** is a word of crucial importance here. It

is the ability to identify the context of a task that allows experts to know which solution is just right for the current problem. They will instinctively know which alternative is likely to only make matters worse.

Inexperienced practitioners of a craft (the workshop novices of olden days) are pleased when someone provides them with clear-cut instructions, especially if they've heard applying these instructions will lead to certain success. Remember what I said earlier about those agile prophets jumping out of the woodwork? Let's be very clear here: no one can tell you what to do in your specific personal context, regardless of your skill level. There is no other team out there that is the same as yours. No other company is a carbon copy of yours. There is no project that is identical to yours. Practices that have worked out great for other projects, might explode in your face. Their context is probably just not right for your specific situation. You need to adapt these successful practices to fit your context. Having someone knowledgeable around to help identify the processes that need customizing is highly valuable, but very rare.

Skill Level/Mental Function	**Novice**	**Advanced Beginner**	**Competence**	**Proficient**	**Expert**
Recollection	Non-Situational	Situational	Situational	Situational	Situational
Recognition	Decomposed	Decomposed	Holistic	Holistic	Holistic
Decision	Analytical	Analytical	Analytical	Intuitive	Intuitive
Awareness	Monitoring	Monitoring	Monitoring	Monitoring	Absorbed

Pragmatism and craftsmanship closely relate. Most people that climb to higher levels of mastery tend to follow more pragmatic approaches. It is my personal belief that an eclectic combination of practices is the way to go to achieve better performance on your current project. It boils down to: *"Don't just do things because some consultant or book says you should do them"*. Always be mindful of your context, and apply those practises that bring value. In order to know which practices are helpful, it is important to understand where they originated from, and the problems they aimed to solve. **Understand the WHY, before worrying about the HOW!**

> Always consider the context. ~Andy Hunt, Pragmatic Thinking and Learning: Refactor Your Wetware

I would advise you to also look for contextual knowledge in the business domain of your clients. If you understand why your client wants something, you are often able to suggest a more valuable alternative to the task they just asked you to complete. You could respond to their request with: *"It would be very expensive to build your application that way. But if I understand your need correctly, doing XYZ might also solve your issue at a much lower cost."*

If you find yourself on a project that is more likely to jump onto the next *hype train* that rolls into town instead of looking for ways to improve your current approach, remember that eventually you will run into serious problems. It is then up to you and your peers to try and make the best of the situation. You could try to convince your management to improve the existing processes, instead of restructuring everything because it is the trendy thing to do. This way, you can steer your project towards the path of pragmatism and continuous improvement. If that fails miserably, you always have the last resort of hauling ass and looking for a less whimsical environment.

I read this article and all I got was this lousy insight

The most important lesson to take away from this article is that **context matters**. External advisors and consultants might have a very deep understanding of a specific process, but they have little knowledge of your team, company, and the issue you are trying to solve. If you want to make your project operate more efficiently, or get out of the mess you've gotten yourself into, **you and your team** will have to do most of the heavy lifting.

Adding experienced and skilled people to your team can help you gather new knowledge and practices. These people are invaluable as they have seen what works and what does not in different environments. If these people stick around on your project for a significant time, and are actively involved in your process, they might uncover some inefficiencies or *'weirdness'* in your approach. Team members that have been on the project for a long time can likely tell you why your team is currently doing things the way you are. These senior members can also provide you with knowledge of the business processes of your clients, and the history of your project. Information such as *"We tried to switch to a different server technology five years ago, but in the end the change did not go through because the client has very specific privacy requirements"* can help you narrow down which of the alternatives you are thinking of is actually viable.

When you are the experienced person to join a team, be mindful of your own biases. Remember that the things that have worked well for you in the past might not work in your new context. You need both a fresh set of ideas, and the knowledge of the people who have been on the team for a long time. The great paradox here is that the persons who have been on a team for a long time are usually the least likely to welcome change. They are also the ones with the most valuable knowledge to make the new idea work. When you try to introduce a new approach to a project, be sure to include these people in your quest.

If adding experienced practitioners to your team is not an option for you, employing consultants on a short-term basis or following a training seminar might be more viable. Just remember that if you do choose this option, it is up to you to understand the approaches they are trying to sell and the problems those approaches can solve. These consults and training seminars are useful to change your perspective of things, and to show you possible alternatives to your way of working. Armed with this knowledge, you can go back to your daily tasks and be mindful of inefficiencies. Maybe that *"Test Driven Development"* training you took a few months ago was on to something, and maybe applying those practices will help your team produce less bugs and miss less deadlines.

You can always start trying a new approach. If you continually evaluate if it is helping your team, or rather slowing it down in the long run, you will know whether to keep the process or not. The core values of the The Manifesto for Agile Software Development, and it's twelve principles of agile software are a great point of reference if you are in doubt of what to try next. But remember they are **references and advise**, not the law.

Listen to your team, gather ideas from skilled people (internal and external), and be on the lookout for improvements to your process. But most of all, care deeply about what you are doing.

TL;DR

- *"doing Agile"* will not fix all your problems
- just because some company is successful while doing something, does not mean you will be successful if you blindly copy their approach, Consider the context.
- experienced people use the tools and processes that make sense to **their project**, **their team**, and **their corporate environment**

- time is your friend
- If you aim to master your chosen craft, be pragmatic, motivated, and keep exploring!
- If you are in a leadership position, **attract and hire motivated people who care about their craft**. They will bring greater value to your project than any process or management hype could hope to ever do.
- Listen to your team, gather ideas from skilled people (internal and external), be on the lookout for improvements to your process.

Appendix

Metadata

Property	Value
Originally Published on:	2020-05-04
Author:	Stijn C. Dejongh
Original title:	Pragmatism and Software Craftsmanship
Article source:	https://github.com/sddevelopment-be/penguin-programming
Licensed under:	EUPL

References and Further reading

If this overview of agile software development and craftsmanship has inspired you to read more about the subject, I invite you to take a look at the books and video material I based this article on.

author	url	title	publisher
Beck, K; et al	agilemanifesto.org	Manifesto for Agile Software Development	Ward Cunningham
Dreyfus, Stuart E; Dreyfus, Hubert L	http://www.dtic.mil	A Five-Stage Model of the Mental Activities Involved in Directed Skill Acquisition	Storming Media
Hunt, A	pragprog.com	Pragmatic Thinking and Learning: Refactor Your 'wetware'	Pragmatic Bookshelf
Hunt, A; Thomas, D	pragprog.com	The Pragmatic Programmer, your journey to mastery	Addison Wesley/Pragmatic Bookshelf
Hunt, A; Subramaniam, V	pragprog.com	Practices of an Agile Developer	Pragmatic Bookshelf
Hoover, D; Oshineye, A	oreilly.com/library	Apprenticeship Patterns: Guidance for	O'Reilly Media, Inc

		the Aspiring Software Craftsman	
van Solingen, R	talk on youtube	De Kracht van Agile (8 ankerpunten voor de praktijk)	Agile 2019 talks - distributed on youtube.com

Concepts

`/2_Productivity/Concepts`

Return on investment

Context

There is great difficulty is in expressing the "Value" and "Cost" of activities and projects. In a simplified business environment, these terms are usually substituted by "money earned / money invested". While this is a useful metric for sure, I feel that it lacks some non-monetary gains. There are those that try to calculate the value of everything by converting it into an amount of money. While there is value in this, I find it a difficult exercise to align with my personal ethics.

A question to ask here is: *"How would you value a human life? How would you value an improvement to someone's self-image or daily struggles?"*

We shan't go into this much further, as we would deviate from the main point I wish to transfer and dive too deep into philosophical debate. The point I wish to make here is: there is often more to be gained than just money. And even if we would be able to quantify those " intangible" gains, we can also not predict the future. Which means that the definition of "value" will always be a subjective concept.

Even though this subjectivity is not something we can get rid of, it does not mean an empirical approach on top of this holds no value. Statistics as a field are a prime example of this statement. Their usefulness is not in the absolute numbers they provide, but by offering a formalized way of comparing different situations and contexts. As we all know, they are mere tools to help people and organizations define their strategy. Their usefulness is limited to how we interpret them, and what actions we take based upon them.

It is the same for a metric such as ROI. Its use is to be one of many metrics that can help guide you to make more informed decisions. But as with all metrics, it is up to you to interpret the data and make decisions. If you go one step further in this reasoning: it is the person who defines what "value" and "cost" means that decides what to include in their strategic decision-making process. This in turn means that the metric itself is only as good as the contextual knowledge and expertise of the person (or group) defining the input.

Description

ROI or "Return on investment" is a very useful metric for strategic planning. The idea is to maximize for value over time. This helps you to stay focused on the activities or task that bring the most value to your organization/project/customer, or even your own life.

The basic formula to calculate ROI is: **Value gained / Cost of activity**. I will leave the exact formula's to go from this calculation into percentage based metrics, etc.

Embrace the fluffiness of this metric, and focus on what is important to you and your context. Try and find a way to go from a qualitative to a quantitative evaluation of tasks. Once you and your context agree on those, the ROI metric becomes a useful tool for tracking the impact of the work you do to reach your goals. It is to be used as a tool for comparison to a previous period in your path. This way, you can see if you are progressing in the way you wish to progress or if you need to adjust your course.

Measuring success

- Your reports and analysis documents contain an ROI indication
- You think about cost/benefit ratio's while working

References

Item	Description	Action
Good Enough Code	A programmer mindset pattern that aims to match code quality recuirements to intended use	Read the GEC pattern in the Programming section and reflect on how it relates to this.

Practices

`/2_Productivity/Practices`

Free your mind: the external memory

Context

Human memory is extremely lossy. We are better equipped for creative, constructive thinking than for storing factual information. Focus is easily disrupted. It makes practical and economic sense to try and reduce the impact of *"forgetfulness"* and being *"pulled out of your flow"*. Try and find a way to free up your headspace and thinking power for the endeavors that actually matter.

Drivers

- Context switches hurt your productivity
- Modern systems and activity are too large to pit in your brain completely
- The more you have to keep in memory, the more likely you will forget something
- Having a mental task list becomes exhausting after a while
- We wish to be able to easily report progress when queried

- Hand-overs take a lot of time and effort
- Computers are great at remembering stuff

Solution

- Use a technological or physical aid to keep track of your ideas and notes
- Make sure you **trust** your external brain, in order to free head-space
- Revisit your notes regularly
- Favor text-based formats, as they are easier to version, maintain and port
- Whatever system you use, make sure it is easily accessible, and non-disruptive to your primary focus

Examples

There are many online (free and paid) systems that offer you a way to store your thoughts. Some of them are listed below in the *references* section.

You can use a simple text-based system to keep track of your thoughts, or ToDo items. I personally prefer using these text-based systems as they allow for easier cross-platform portability, akd avoid you being locked in to a single vendor solution. This entire knowledge base can be seen as one big "external brain".

Personal testimonials / opinions

Stijn's developer logs

> [!TIP] One of my favorite ways of note-taking while working on a software project is to add a `dev_notes` directory to my codebase. For short-lived projects or changes, I tend to add this file to the `.gitignore` configuration of my repository as to not muck up the workspace of my colleagues. For more long-term or collaborative projects, consider creating a dedicated repository to host all of your developer notes. You can get creative with `symbolic links` to make these folders show up in your codebase regardless of their physical location.

References

Item	Description
Creating a personal wiki	Asian efficiency: Creating a personal wiki
notion.so	Online note taking
Create a fersonal wiki using MS OneNote	Article by I. Humphrey n using OneNote as an external brain
NextCloud Personal data server	DIY data and notes storing solution
TODO.txt format	An open, text-based format for your TODO files
Trambu	My owm todo.txt inspired task-management application
Emacs org mode	A major mode for GNU Emacs, aimed at organizing your thoughts

Organize your workflow

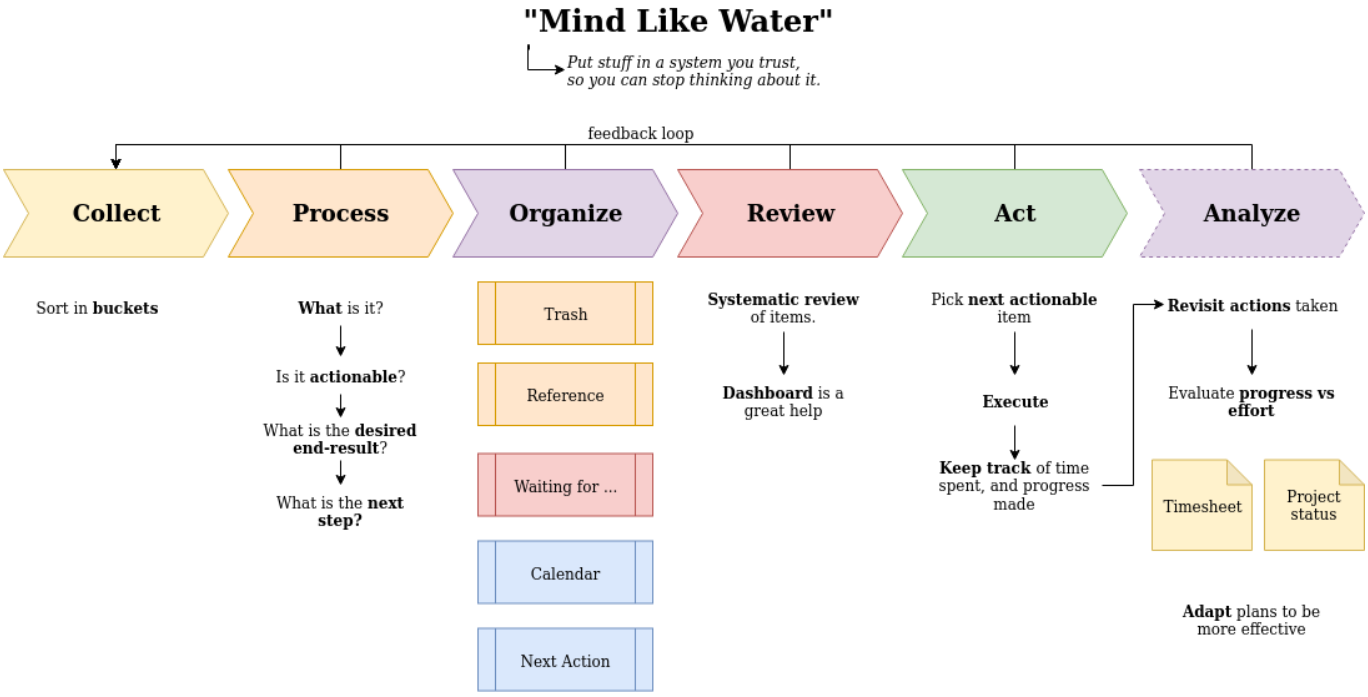
Context

Drivers

- You want to finalize your tasks and "get things done"
- People are often distracted by their own thoughts
- Multitasking is hard. Our brains work better if we can compartmentalize between different modes of thinking
- Our memories are very lossy. We tend to forget about things all the time
- Standard To-Do lists have no readily available means of prioritizing items
- Having too many things on our mind stresses us out

Solution

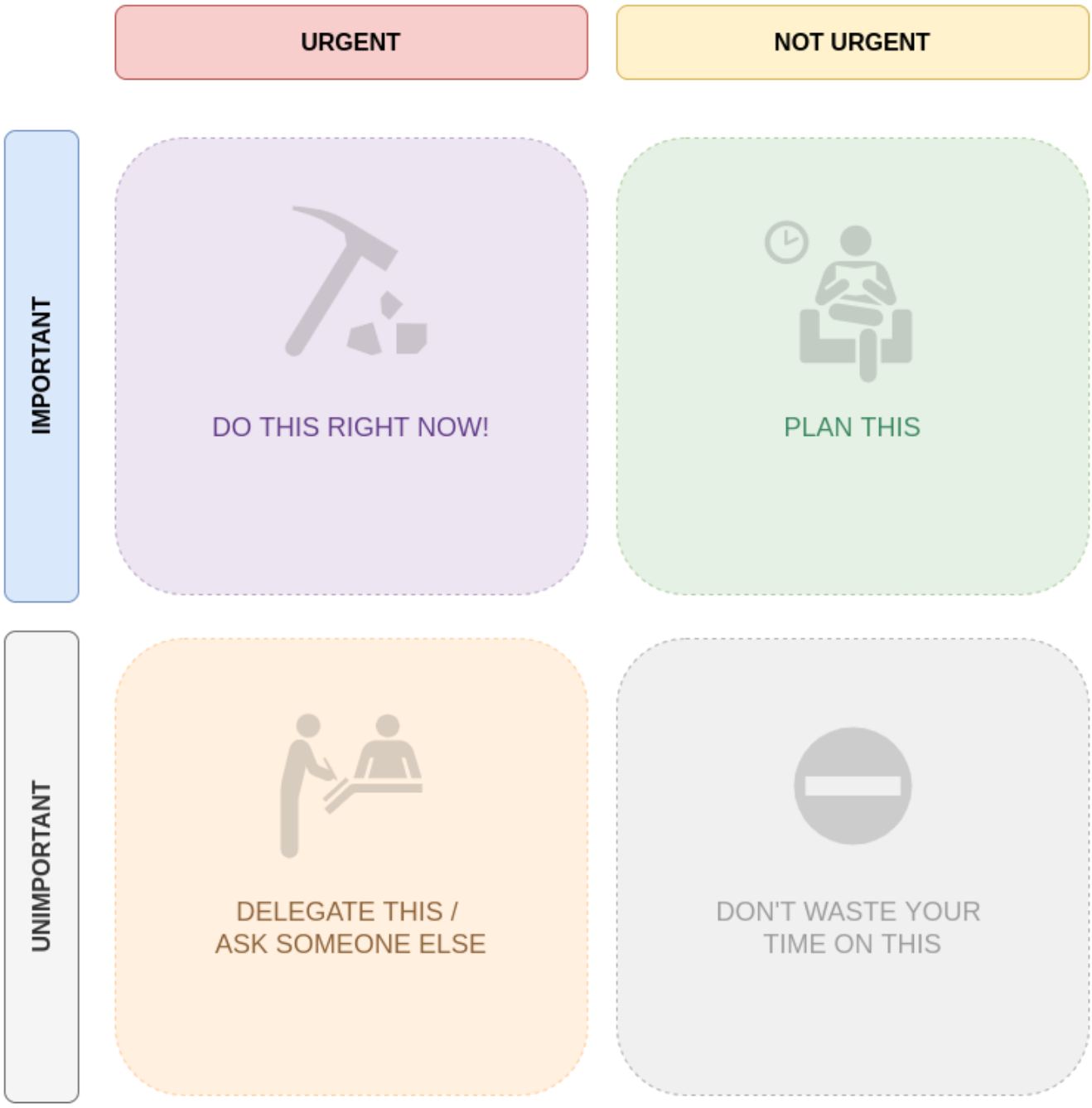
Use a personalozed productivity flow, using whatever tool you feel comfortable in. For some, this will be a pen-and-paper system. Others prefer to leverage their digital prowess and lifestyle to have theis productivity tool available all the time[^1]. One of the most popular personal productivity methods is called the *"Getting things done"* method[^2].



Whatever method you choose, make sure to:

- Have a specific location to dump out your thoughts
- You should **trust** your tools. Peace of mind only happens when you are convinced your ideas and todos are stored reliably
- Regularly review your braindump tool and curate it. Some of your ideas will be crappy. Get rid of them.
- Categorize and Prioritize your thoughts **after** you have collected them
    - The Eisenhouwer Matrix is an excellent technique to do this
    - differentiate between "things to remember" and "things that require action"
    - assign "due dates" if able

Eisenhouwer priority Matrix



Organize your action items according to:

- their **importance**: actions can be either `important` or `unimportant`
- their **urgency**: these are either `urgent` or `not urgent`

This leaves you with a nice 2-by-2 grid in which to place your actionable items[^3]. Each grid section corresponds to a specific way to handle the action items contained within:

- **eliminate of ignore** the things that are `unimportant` and `not urgent`. Because *Who cares abut these anyway?* The items in this quadrant are better known as "distractions" or "busy work".
- **delegate** the `unimportant` and `urgent`stuff. Something should be done about them fast, but you probably don't need to be the person working on them.

- **plan / schedule** the items that are `important` and `not urgent`. These are the things that you would really like to do, but never seem to get around to doing. Setting a specific date on which you will act upon these items will help you get them done eventually. Just make sure to stick to your schedule.
- **Do these NOW!**: `important` and `urgent`... what are you waiting for? Start working on these immediately. Just make sure these things are important to YOU and you are not being swayed by someone else's urgency.

Examples

Personal productivity flow using MS online tooling

In recent years, microsoft has pivotted towards providing office-as-a-service applications. Most if these can be easily incorporated into your personal workflow. Having everything online makes it easier to be productive, without losing your mobility.

References

Item	Description
Getting things done - David Allen	Link to book (amazon)
NextCloud	A self-hosted personal filen organizer, and productivity platform.
Eisenhouwer matrix	Article on ProductPlan.com discurring the Eisenhouwer matrix
Notion.so	An easy to use, external notebook that allows for limited automation and offering a wide range of customizability and plugins
Trello	A simple Kanban board to track your main tasks and their status

[^1]: Unless of course your computers die and your internet connection goes on hiatus
[^2]: From the book 'The 7 Habits of Highly Effective People: Powerful Lessons in Personal Change' by Stephen R. Covey
[^3]: Mathematicians (and computer programmers) call this a *"Matrix"*. We are sorry to disappoint you if you expected leather clad martial artists to help you out with setting your priorities.

OPERAs method

Context

> A mental model to work towards task progress

More often than not, people tend to have a big list of things they would like to get around to doing. Finding out how to make progress towards a specific goal can be quite challenging. In order to help with this endeavor, the OPERAS mental model can be useful. This is a general purpose pattern to frame progress towards a specific goal by using the *"divide and conquer"* technique and a clear checklist-style plan of attack.

The core idea is to give pattern users a mental map of the different actions that are required to complete a 'milestone'.

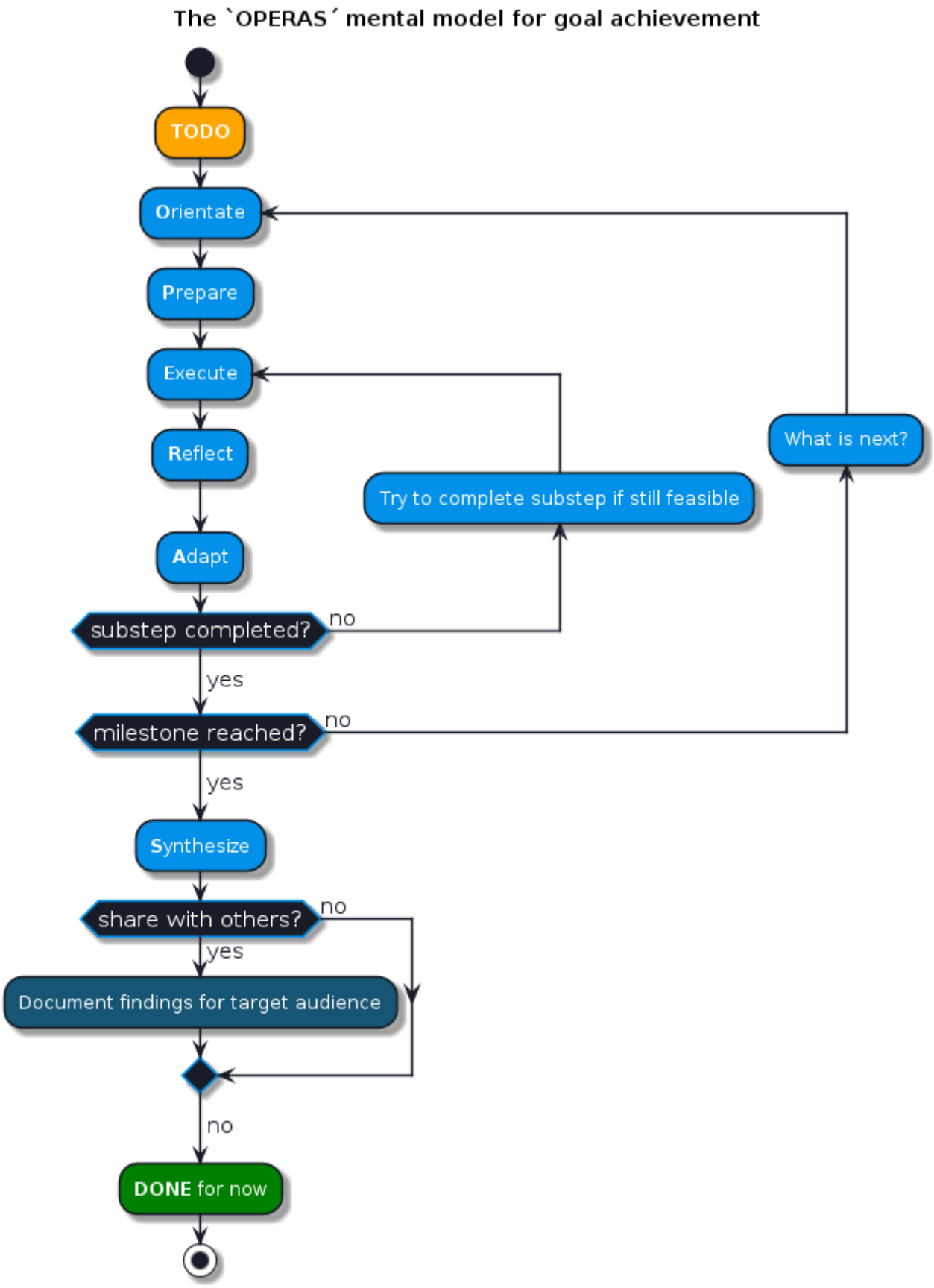
Drivers

- you want to keep tabs on the status of various goals and objectives, by using the step names as status indicators
- having a clear idea of progress made, when combined with the KPI pattern
- split big goals into bite-sized chunks
- stay committed to your goals without feeling like you are drowning in a never ending cycle of irrelevant 'side quests

Solution

The OPERAS mental model is a description of the flow you go through when advancing a task from "TODO" status to "DONE". A task can be of any size, and can range from 'build a house' to 'do the dishes'.

- define a goal you wish to work towards
    - It is best to describe this goal in a measurable way, so tracking progress is easier in the future
- when working towards a goal, starting from a 'to do' state, you perform a series of steps in order
    - for some processes, a few of these steps could require very minimal effort.
- the acronym 'OPERAS' stands for the main steps in this model's flow:
    - Orientate
        - Prepare
        - Execute
        - Reflect
        - Adapt
        - Synthesize



- ▶ 1. Orientate
- ▶ 2. Prepare
- ▶ 3. Execute
- ▶ 4. Reflect
- ▶ 5. Adapt

▶ 6. Synthesize

Measuring success

- While working towards a goal, you can measure your progress in a numerical manner, if the goal so allows.
- Are you reaching more milestones?
- Do you feel like you are more productive?
- Do you have an enhanced sense of purpose and/or productivity?

References

Item	Description
most agile frameworks	the concepts of `increments`, `reflection` and `adaptation`
A. Hunts GROWS method for digital transformation	A well thought-out phased improvement track for knowledge workers and software professionals
KPIs and OKRs	useful to measure progress
the "external memory" pattern	keep track of your goals, tasks, and progress
the "dream game" pattern by Matt Barnes	A nice way to figure out what you are aiming to achieve

Engaging retrospectives

Context

> A retrospective, generally, is a look back at events that took place, or works that were produced, in the past.

Hosting a useful retrospective session is quite difficult. You aim to have a group share their frustrations, obstacles, and share knowledge about what went well. One of the risks of having a static retro format, are that teams lose interest in the sessions, and are less engaged during the meeting. We aim to avoid this staleness by varying the "theme" of the retro, while keeping the general structure the same.

Every group can employ slight variations on the approach used, based on a similar template. The aim of these alternating retro formats is to stimulate constructive communication between team members.While the team lead or management are present during the discussions, and have the final decision on actions proposed, they should refrain from taking an authoritative position during the retro.

Drivers

- You are helping a team or project being in a rut, going through the motions without engaging or following-up on issues and frustrations.
- We can only start improving the situation once we know what is going on
- People perform better when they feel involved in their work
- Most people are intimidated by authority

- People do not like to feel ridiculous and want to be taken seriously

Enablers

- Team/Management is willing to try, and dedicate time to improve their process
- Trust in the intentions and skills of the facilitator or person taking the initiative
- Team members feel safe to share their opinions without fear of reprisals

Deterrents

- The group has no motivation to experiment with alternative ways of working
- You are operating in a low trust, high conflict environment
- You can not invest the time to be a driving force of change

Solution

- Gather the team, and openly discuss what it was like to work together over the past X weeks.
- Pocus on making sure every voice is heard, and the discussion is not dominated by a couple of strong-willed individuals

> "A plan or framework is very useful, as you need something to adapt and deviate from as you gain experience." ~ Paraphrased quote by Rini van Solingen

The sections below are intended as a starting point for hosting your own retrospective sessions. You are obviously welcome, and even encouraged, to adapt them to fit your group's specific needs. We do recommend giving the described structure a try first, to get a feel for the practice before changing it.

Retro participant roles

Facilitator

Guards the integrity of the retro. He/she is the timekeeper, and the lord-and-master of the retro. During the retro, this person is allowed to overrule any participant or tell them to shut up (Even if they are their superior in the group hierarchy).

> From personal experience, the person executing this role is more effective when not participating in discussions or votes. The downside of sitting out the session is made up for by rotating this role.

Participant

Joins the retro, raises issues (tickets/stickies, etc), votes on discussion points, engages in constructive dialogue.

Observer

> "You can sit in the room/meeting/call, but you are not to speak"

Is invited to listen to the retro, but does not participate. If an observer does not stick to their accepted role, the facilitator should give them the choice between switching to a participator role, or leaving the meeting.

Scribe

can be combined with any other role

Responsible for taking notes on discussions and actions taken.Personal experience teaches that this is best combined with the roles " Facilitator" or "Observer".

Retro structure

When using a specific retro theme, feel free to tell a small story (1-2 minute fluff). Bonus points for everyone that looks amused.Verify everyone understands the goals and outline of the retro.

- ▶ 1. Context / Set the stage
- ▶ 2. Warm up (~5min preparation)
- ▶ 3. Personal preparation of discussion topics
- ▶ 4. Round-table presentation of possible topics
- ▶ 5. Reducing ticket count / Grouping
- ▶ 6. Voting
- ▶ 7. Discussion and consensus-based action definition
- ▶ 8. Wrap up
- ▶ 9. Checkout
- ▶ 10. Follow-up

Measuring success

- The group is engaged during the sessions
- Multiple voices are heard
- You are able to stay within the assigned timebox most times
- The actions decided upon during the retro are executed

References

Item	Description	Action
FunRetrospectives.com	Paulo Caroli && al.	Get inspired by these plug-and-play mini-themes and icebreaker exercises
Agile retrospective ideas to keep your team engaged	Michael de le Maza	
40 ideas to spice up your retrospective	Ralph van Roosmalen	
11 Ideas to Spice up Your Retrospective	Barry Overeem	
Sprint retrospectives ideas	Various authors	

3_People_Skills

/3_People_Skills

> [!WARNING] **This is not a bingo chart.** Unlike when playing Pokémon, you will not get a special prize for getting all the patterns in this publication into your work or life. You are encouraged to be eclectic, and only use the practices that resonate with you. Feel free to change them as you see fit, after all: Who are we to tell you what to do?

No one is an island

When reading stories or career advice for people in the technical sector on social media, much content is dedicated to technical excellence and workflow processes.

You might be familiar with phrases such as *"You won't really be a developer unless you learn such and such framework"*, *"All good developers have read this and that book"*, *"You really need to know these and those concepts"*.
While most of this is true (to a certain extent), most people that work in the tech industry tend to consistently undervalue the importance of being able to express yourself. There is a lot of value in generally getting along with the people you interact with on a daily basis.

> Having great ideas gets you nowhere fast, if you are unable to communicate your knowledge to others.

The stereotype of the computer geek sitting in a basement and only interacting with people through some sort of ticketing system can be quite accurate to some extent. Personally, I feel best when I have the sense of moving forward and collaborating on tackling a challenge together with my peers.

> No man is an island
>
> ~ John Donne

Articles

/3_People_Skills/Articles

A reflection on communication failures

> Originally published on 2019-08-10 by Stijn Dejongh (personal blog)

As one of my college professors used to say: I hate to come over as a "sage on a stage", but over the last few months I have come to learn a whole lot of intangibles. Some advice that was given to me in the past by more experienced former colleagues has started to make a lot more sense, and I felt the need to write it down and share some realisations, advice, and experiences I have had.

When I read stories or career advice for people in the technical sector on social media, I usually feel that way too much attention is given to technical excellence. Phrases such as "You won't really be a developer unless you learn such and such framework", "All good developers have read this and that book", "You really need to know these and those concepts", are very common advice given in discussions about doing well in a technical profession.
While most of this is true to an extent, most people that are not active in a professional context (and even

some that are) tend to actively undervalue the importance of being able to express yourself and to generally get along to some extent with the people you run into on a daily basis.

The main argument here is: **It does not matter how correct and technically skilled you are, if you are unable to communicate your knowledge to other people.** The stereotype of the computer geek sitting in a basement and only interacting with people through some sort of ticketing system can be quite accurate to some extent. Personally, I feel best when I have the sense of moving forward and collaborating on tackling a challenge together with my peers.

My previous job was my first professional experience in a real-world software development team. I joined the team as an absolute beginner. The terms *"junior"* or *"young graduate"* used in the industry were very much applicable to me. As I was inexperienced, I tended to spent the first few months of my employment mostly on the sidelines, feeling as if my opinions were quite worthless when compared to those of my senior colleagues. Most of the time, I tried to absorb as much knowledge from them as I could and being in awe with how competent they all were. I had the considerable benefit that they were all friendly and open people who actively encouraged me to speak up and voice my opinions. I believe it is quite common for someone starting their professional career to hold the believe that the workflow of their first team is *"the way things are supposed to be done in the real world"*. I know I felt that way at least.

Over the years that followed, my technical knowledge and professional network grew considerably as colleagues came and went. I began to realise that there are alternative ways of doing things. Other approaches, workflows, and technologies seemed to be just as viable as the way we tended to go about our business. Some even seemed to be more desirable. This is when I started to want to experience other things than what I was used to, and grow more. I ended up changing to a different company. Before I started working at my current project, I thought I was as prepared as could be for the change that was going to happen. I mentally prepared myself that the codebase would be different, that the technology would be different, and that the team worked would be different.

The thing I was not prepared for was that the people would also be very different. I spent the first months at my new job looking for my place in the team. I wanted to fit in, and make a positive impact. I no longer felt like the inexperienced junior, and I believed that I had valuable information to share. I just did not know how, and when to speak. The other thing I was unprepared for was how fast I would pick up on small things that were not quite right. At my first job, it took me years to realize that the way we did things was less than perfect and that we could make improvement on almost all aspects of our day-to-day regimen. At my current job, it only took me a few weeks to find possible improvements. I started to point some of these things out. Since I was used to being quite blunt and factual in the way I said things, this did not come over particularly well with some of my colleagues. Their seeming inability to see things from my perspective left me frustrated with these colleagues for a while. I wondered why they would not give my ideas a chance, and was borderline angry with their responses to my suggestions.

I then found a video online on 'non-violent communication'. Because of the struggles I was having communicating my ideas at work, I gave the very lengthy video a shot and watched it. At this point, I have to admit that it took me a viewing sessions or two to get through the whole thing, and that I skipped most of mister Rosenberg's songs. Even so, I picked up on a few core ideas that I think are quite valuable. He spoke on actively listening to what the other person is trying to say, in stead of getting all caught up in the words that they use. I also realized that at work I was communicating in a way that matched my previous team, and not in a way that fitted the people in my new environment.

After reading through the book, I think the most valuable lessons were that you want to make sure that if you want your thoughts and emotions to be heard, it is best that your communication comes over as friendly, welcoming, and non-demanding. I realized that the way I tend to say things could come over as harsh and that it could make people feel as if I was criticizing their work or value to the team. Obviously that was never my intention. The book also made me realise that when others get annoyed or angry, or when they seemingly criticize me, my first reaction would be to defend myself and respond in force. This tends to escalate situations, or at the very least ensure that both parties are actually holding intermittent monologues, rather than having a conversation.

The main things I took away from reading about a non-violent form of communication are:

- Try to actively listen to what other people are saying, in stead of just hearing the words
- People are not insulting or threatening you, but are trying to convey that they are not happy about the present situation
- When you find yourself unable to actively listen to what others are saying, it is usually because you have some sort of unmet need of your own
- Most people are just trying to go about their business in a way that meets their own needs and expectations
- Your point of view can drastically change the way you think about another person

There certainly is a lot more to it, but these realizations have certainly made the biggest change to how I try to go about interacting with other people. I can not say I always succeed in having valuable conversations, but being aware of them have certainly made me feel a lot more at ease in how I receive feedback and how I deal with people that are clearly upset about something.

I want to share more realisations on different subjects, but I feel as if this post has grown to be lengthy enough as it is. I will end it here, and maybe write more of these kind of posts.

Concepts

`/3_People_Skills/Concepts`

Leadership

`/3_People_Skills/Concepts/Leadership`

Lewin's Leadership Styles

Description

In 1939, a group of researchers led by psychologist Kurt Lewin set out to identify different styles of leadership. While further research has identified more specific types of leadership, this early study was very influential and established three major leadership styles. In the study, schoolchildren were assigned to one of three groups with an authoritarian, democratic or laissez-fair leader. The children were then led in an arts and crafts project while researchers observed the behavior of children in response to the different styles of leadership.

Authoritarian Leadership (Autocratic)

Authoritarian leaders, also known as autocratic leaders, provide clear expectations for what needs to be done, when it should be done, and how it should be done. There is also a clear division between the leader

and the followers. Authoritarian leaders make decisions independently with little or no input from the rest of the group.

> Researchers found that decision-making was less creative under authoritarian leadership. Lewin also found that it is more difficult to move from an authoritarian style to a democratic style than vice versa. Abuse of this style is usually viewed as controlling, bossy, and dictatorial.

Participative Leadership (Democratic)

Participative leaders encourage group members to participate, but retain the final say over the decision-making process. Group members feel engaged in the process and are more motivated and creative.

> Lewin's study found that participative leadership, also known as democratic leadership, is generally the most effective leadership style. Democratic leaders offer guidance to group members, but they also participate in the group and allow input from other group members. In Lewin's study, children in this group were less productive than the members of the authoritarian group, but their contributions were of a much higher quality.

Delegative (Laissez-Faire) Leadership

Delegative leaders offer little or no guidance to group members and leave decision-making up to group members. While this style can be effective in situations where group members are highly qualified in an area of expertise, it often leads to poorly defined roles and a lack of motivation.

> Researchers found that children under delegative leadership, also known as laissez-fair leadership, were the least productive of all three groups. The children in this group also made more demands on the leader, showed little cooperation and were unable to work independently.

Usage

References, Related Patterns and Resources

Item	Description	Action
The Leadership Style Matrix	Jeff Doolittle	
Definition of Leadership	Kurt Lewin && al.	
Patterns of aggressive behavior in experimentally created social climates.	Lewin, K., Lippit, R. and White, R.K. (1939)	

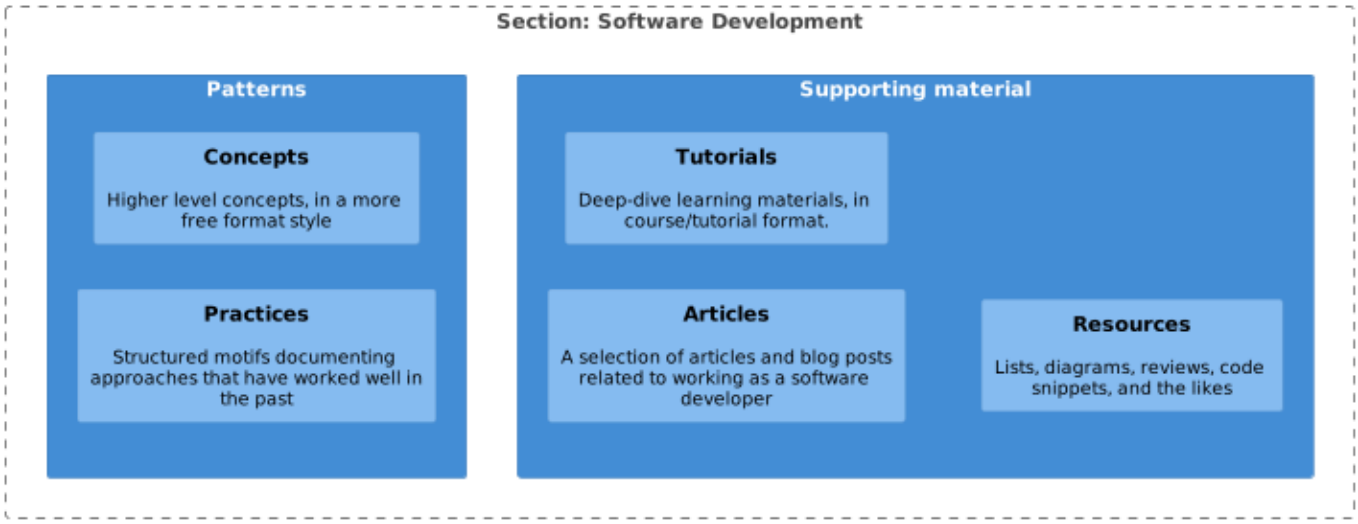
Practices

`/3_People_Skills/Practices`

4_Software_development

`/4_Software_development`

Patterns for programmers && techies



Plenty of source material exists out there for developers. Admittedly, a lot of those books are more factual than this one will ever be.

During my time coaching less experienced people, I noticed that the sheer volume of good information can be quite overwhelming for a beginner in the field. So in good old engineering fashion, I am combating the overabundance of good standards by creating yet another one. My hope is that this section will give you an introduction on what it takes to be good at our job. If all goes well, it should encourage you to dig deeper into more specialized material, which can be found in the Appendix section.

> [!WARNING] **This is not a bingo chart.** Unlike when playing Pokémon, you will not get a special prize for getting all the patterns in this publication into your work or life. You are encouraged to be eclectic, and only use the practices that resonate with you. Feel free to change them as you see fit, after all: Who are we to tell you what to do?

Code Samples

This section includes a variety of programming patterns that have been collected, and put to use in both personal and professional projects. In order to make learning them a bit easier, we have included code samples to these patterns. They can help you understand the approaches described in the patterns and offer a great way of getting some practical experience with them. The code included in the patterns can be written in a variety of languages, but the default language is Oracle Java.

Technology	Download link
Java Development Kit	Oracle website

Articles

`/4_Software_development/Articles`

Avoid Cargo Cult database design

> [!INFO] Originally published on 2019-08-28, by Stijn Dejongh as `Beware of hibernate killing your design`

Shooting yourself in the foot because of Hibernate "Best practices"

A few years ago, one of my colleagues had some rather shocking things to say about the use of hibernate in big software projects.

> "We should leave hibernate and database modelling to senior developers. A lot of projects I have worked on have been ruined by inexperienced people writing Database interaction code."

At that time we were working on enhancing the performance of an application that was pestered with N+1 problems due to lackluster modeling and database design.

We are now a few years later, and a recent interaction with a fellow developer made me realize that performance problems due to how easy hibernate makes it to links database tables is only part of the issue.

To understand what I mean with this, I will start with a simple modeling example, in which we do not think about database interaction at all.

A simple object modeling example

We start with a large monolithic system that contains multiple simple existing domains. Amongst others, the system has parts that are responsible for managing a corporate fleet. The system also has a separate part that is used to manage shoe stores. The example is a bit extreme, as these sub domain are clearly miles apart, but bear with me. It is abundantly clear that the Car system has little to do with the ShoeStore system. We have followed a layered approach during our coding, and since we don't want to expose too much of our internal workings, the Cars system is controller through the "CarApplicationService" that contains some access control logic. Likewise, the ShoeStore domain follows the same pattern.

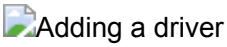
 Initial model

All is fine and dandy, and desired pathways to our objects are clearly defined. If a class of the Car domain needs to access anything in the ShoeStore domain, it is required to go through the ShoeStoreApplicationService in order to manipulate any Shoe. And even though our system is a large monolith, our structure is quite clean.

In comes the following requirement

> We wish to be able to get a list of all the names and surnames of the drivers of the cars in our fleet.

If we were to naively do this, and just add a "Person" field to the Car class, we would end up with this:

 Adding a driver

Now we have added a pathway that enables us to write code such as:

```
//Stuff
car.getDriver().getShoes().getShoeStore().getName();
//More stuff
```

In this example it is apparant that this is not the way to go, that there is too much coupling, and that we need to rethink our model. We might come up with a different solution for referencing person names.

In comes hibernate

There are a lot of cool ORM frameworks out there that allow you to be more productive, and not worry to much about database stuff. **Hibernate** for instance, enables you to annotate fields with code such as this:

```
@Column(name = "PERSON_SURNAME")
private String surName;
```

This enables us to not worry to much about the specific database queries that will be executed, or even which specific database technology we use. And it is very easy to just add these annotation to our domain classes and know that hibernate has our backs and handles all the messy database interaction magic. One of such easy features, is that it allows us to hide join opperations on the database by modeling database links with foreign keys as if they were internal objects. Again, allowing us to code without worrying about the database stuff too much. We tend to write code like this without thinking much of it:

```
@ManyToOne
@JoinColumn(name ="FK_PERSON_ID")
private Person person;
```

This helps ensure us that the data sent to (or retrieved from) the database will always be valid, as otherwise we will run into exceptions and stacktraces. Because of the ease of use hibernate gives us, most developers consider it good practise to encode database relationships in their code in this way.

So what's the fuzz?

Everyone knows of the design practices of separating your database entities from your domain code. But in practice, this usually results in developers having to make a lot of changes, in a lot of different classes when a new attribute is added to a class. Most programmers in the business agree that while writing DAOs and domain classes separately is a nice idea, the gain does not weigh up to the effort for most projects. Since I value efficiency and pragmatism, I tend to somewhat share this idea.

The combination of previous sections is what makes hibernate a somewhat dangerous tool, if you are not paying full attention to the object model you are creating without fully realizing it.

In the java code, there is a huge difference between modeling your database table that contains a foreign key to a person as either a reference field (a `Long`, as per the database field content) or the refered object.

With hibernate magic:

```
@ManyToOne
@JoinColumn(name ="FK_PERSON_ID")
```

```
private Person person;
```

Without hibernate magic:

```
@Column(name ="FK_PERSON_ID")
private Long personId;
```

The second way of writing the java code does not allow you to access the Person class, while the first one does. So my advise is as follows:

> "Do not let notions of hibernate "best practices" make you forget that you could be unintentionally coupling (domain) classes in your code."

and:

> "If you put in a path to another class, expect other people to use it. They might be the ones deviating from the original intentions, but you are the one that made it easy to do."

Concepts

`/4_Software_development/Concepts`

What is Software Architecture?

The nuance between "architecture" and "design" is difficult to grasp. For me one is an extension of the other, but the nuances of the borders of these 'blocks' are elusive.

Definition

In simple words, software architecture is the process of converting software characteristics such as flexibility, scalability, feasibility, re-usability, and security into a structured solution that meets the technical and the business expectations. This definition leads us to ask about the characteristics of a software that can affect a software architecture design. There is a long list of characteristics which mainly represent the business or the operational requirements, in addition to the technical requirements.

Characteristics

As explained, software characteristics describe the requirements and the expectations of a software in operational and technical levels. Thus, when a product owner says they are competing in a rapidly changing markets, and they should adapt their business model quickly. The software should be “extendable, modular and maintainable” if a business deals with urgent requests that need to be completed successfully in the matter of time. As a software architect, you should note that the performance and low fault tolerance, scalability and reliability are your key characteristics. Now, after defining the previous characteristics the business owner tells you that they have a limited budget for that project, another characteristic comes up here which is “the feasibility.”

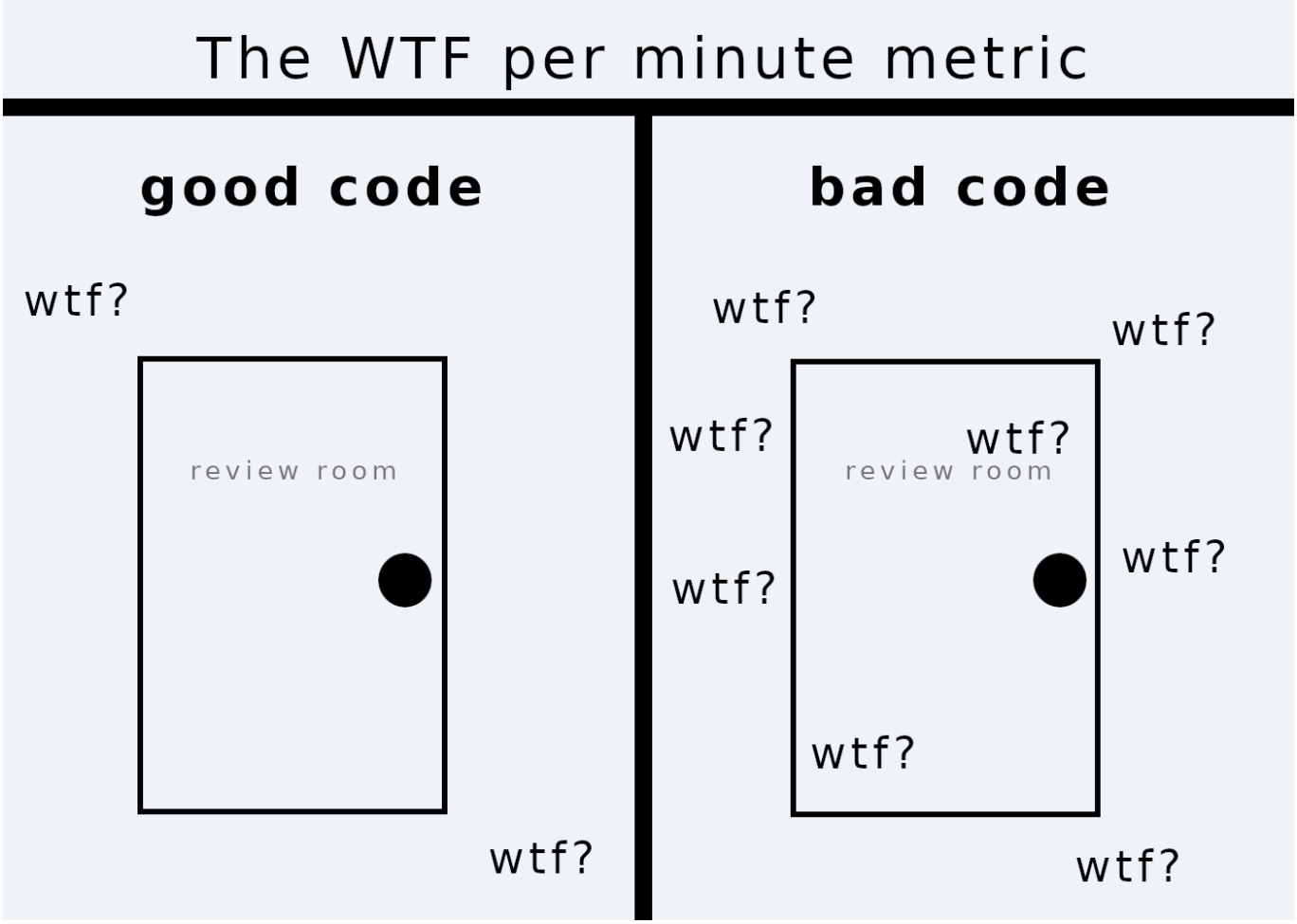
A list of Software characteristics, known as *"quality attributes"* can be found on wikipedia.

Clean Coding

> "Aim to write working, understandable and maintainable code."

Most of us have heard this or similar phrases being uttered by our seniors. It's easy to understand the concept: when you write clean and understandable code, it will be easier to extend and maintain. The big challenge here is in knowing just exactly **how** you write good code. Experienced developer seem to have a gut instinct, allowing them to know when their code is good and when it needs to be cleaned. Over time, you will develop this skill. But when starting out, it helps to know a set of of best practices and see if they apply to the code you have just written.

But it works! That's all that matters, right?



When writing software, it is easy to fall into the trap of thinking that your only priority is making it work. It is true that this is the most important goal of your task. If you have spent some time working on a bigger piece of software, you will probably have noticed that you spent a lot of time reading code that is already there. You might be familiar with a sense of confusion when looking at code that was written by someone else. Even if you mainly work in isolation, this confusion can rear it's head when you revisit code you have written a long time ago. The mean reason this happens is the code being plain **hard to understand**. It is usually not expressive enough, or very verbose.

A good measure to assess the cleanliness of your code enough is to use the infamous *"WTFs per minute metric"*. This metric originates from a well-known cartoon that has been recreated multiple times.

When you write clean code, your colleagues and future self will thank you. As is often the case with projects, along the way your path will change. Things that were once not important at all are suddenly critical to the success of your application. If the piece of code that contains this functionality is messy, it can feel like a trip through the depths of hell to meet the changing requirements. So do yourself a favor, and keep your code understandable at a glance.

> [!TIP] Code is write once, but read very often. People prefer pretty things

Practices

`/4_Software_development/Practices`

Give it a name: Baptizing your code, models, and ideas

> There are only two hard things in Computer Science: cache invalidation and naming things.
>
> ~ Phil Karlton

In a surprising amount of fairy tales, myths, and legends the *"power of naming"* is an ancient magical ability that allows you to control things if you just know how it is really called. Programming is not much different. If the entities and variables you work with have revealing names, a confusing piece of code becomes very clear. This clarity is achieved by simple renaming things to be expressive, a feat most modern IDE's can do for you at little cost.

Context

As a software professional, you spend a lot of time reading code. Some of this code was written by yourself at an earlier point in time, other code is written by others. You notice that functions or parameters have names that do not help you understand their intent.

If the code is significantly harder to read than your average fantasy novel, chances are the names of the `functions`, `parameters`, and `variables` are in need of some attention and nursing.

Drivers

- Holding a lot of different ideas in your brain is **mentally taxing**
- Poorly understood code leads to **lazy changes**, or introduction of **bugs** (aka. Code rot, Broken windows)
- The original author of the code is unlikely to be available to explain their intent
- Holding on to significant mental context makes **interruptions**, or context switches, more expensive
- As humans, we **understand text more easily** if we are given sufficient context, and understand most of the words that are used in the fragment.

Solution

Give variables, methods, and classes a clear and descriptive name.
A good name should:

- be a concept that can be used in natural speech (avoid abbreviations)
- not contain technical concepts

- aim to live in the problem domain, rather than the solution domain

If you are unable to find a good name:

- rethink your model
- browse a dictionary for alternative names
- try and extract part of your concept into something that you are able to baptize
- explain what you are trying to model to someone, and ask their input

Examples

Consider the following code snippets, it is likely you understand the second iteration of the code a lot easier than the first one. Some of you might even recognize it as the 'Bowling game kata' Popularized by Uncle Bob. To a compiler both code snippets are identical. Humans however are not computers, Even though most developers would like them to be.

As humans we understand text fragments, including code, better if we are given sufficient context and if we understand most of the words that are used in the fragment. Good code should allow anyone with a fundamental understanding of the language of choice to understand what is going on at a glance. The older you get, the harder it becomes to keep a large stack of working knowledge in your head. If your code requires you to hold a lot of this knowledge just to be able to understand what is going on, it is probably not very well written.

original version

```
package be.doji.sandbox.kata;

public class Main {

    public static final int MRIG = 21;
    public static final int AMOUNT = 10;
    public static final int MAX = 10;

    private final int[] down = new int[MRIG];
    private int cr = 0;

    public void go(int input) {
        down[cr++] = input;
    }

    public int calcResult() {
        int result = 0;
        int counter = 0;
        for (int i = 0; i < AMOUNT; i++) {
            if (caseOne(counter)) {
                result += 10 + bonusOne(counter);
                counter += 1;
            } else if (caseTwo(counter)) {
                result += sum(counter) + bonusTwo(counter);
                counter += 2;
```

```
            } else {
                result += sum(counter);
                counter += 2;
            }
        }
        return result;
    }

    private boolean caseOne(int in) {
        return down[in] == MAX;
    }

    private boolean caseTwo(int in) {
        return sum(in) == MAX;
    }

    private int sum(int in) {
        return down[in] + down[in + 1];
    }

    private int bonusTwo(int in) {
        return down[in + 2];
    }

    private int bonusOne(int in) {
        return down[in + 1] + down[in + 2];
    }
}
```

with names extracted

```
package be.doji.sandbox.kata;

public class Game {

    public static final int MAXIMUM_ROLL_IN_GAME = 21;
    public static final int AMOUNT_OF_FRAMES_IN_GAME = 10;
    public static final int MAX_PINS_PER_FRAME = 10;

    private final int[] pinsKnockedOver = new int[MAXIMUM_ROLL_IN_GAME];
    private int currentRoll = 0;

    public void roll(int pinsRolledOver) {
        pinsKnockedOver[currentRoll++] = pinsRolledOver;
    }

    public int score() {
        int score = 0;
        int rollCounter = 0;
        for (int frame = 0; frame < AMOUNT_OF_FRAMES_IN_GAME; frame++) {
```

```
            if (isStrike(rollCounter)) {
                score += 10 + strikeBonus(rollCounter);
                rollCounter += 1;
            } else if (isSpare(rollCounter)) {
                score += sumOfPinsKnockedOverInFrame(rollCounter) +
spareBonus(rollCounter);
                rollCounter += 2;
            } else {
                score += sumOfPinsKnockedOverInFrame(rollCounter);
                rollCounter += 2;
            }
        }
        return score;
    }

    private boolean isStrike(int rollCounter) {
        return pinsKnockedOver[rollCounter] == MAX_PINS_PER_FRAME;
    }

    private boolean isSpare(int rollCounter) {
        return sumOfPinsKnockedOverInFrame(rollCounter) ==
MAX_PINS_PER_FRAME;
    }

    private int sumOfPinsKnockedOverInFrame(int rollCounter) {
        return pinsKnockedOver[rollCounter] + pinsKnockedOver[rollCounter
+ 1];
    }

    private int spareBonus(int rollCounter) {
        return pinsKnockedOver[rollCounter + 2];
    }

    private int strikeBonus(int rollCounter) {
        return pinsKnockedOver[rollCounter + 1] +
pinsKnockedOver[rollCounter + 2];
    }
}
```

Write 'Good Enough' Code

Context

A mistake passionate programmers tend to make is to over-design simple things to make them theoretically and aesthetically more beautiful than they need to be at that point in time. In doing so, they often end up spending much more time and mental effort on a piece of software than is needed (or will ever be valuable).

Drivers

- the level of refinement of a codebase should make sense for the problem at hand
- people like to show how clever they are
- readable code is easier to maintain
- thinking about future problems can help mitigate them
- unpredictability of future requirements
- development time costs a LOT of money
- not all code will have a significant lifespan

Solution

- write code that is as well designed as it needs to be at this point in time.
- make sure the code you write at this point in time adheres to the basic principles of clean code and design

- when an idea for a more generic solution comes to mind during your implementation, take note of it and revisit it afterwards
- iteratively enhance the codebase when it makes sense to do so: when tackling a new code challenge, look for reusable components or structural improvements

> [!NOTE] This approach is also referred to as "avoiding gold plating"

Examples

Self-diagnosis

Ask yourself:

- *"Is this code likely to be changed/expanded in the future?"*
- *"Is my design solving an issue that is here NOW, or am I solving an issue that might never happen?"*
- *"If this expected issue occurs in the future, can it be easily fixed at that time"*

Indicators

- Throughput time of changes
- Regression introduced during tasks
- Function point count of changes
- Cyclic complexity
- Readability

Related Resources

Item	Description	Action
Enterprise Quality FizzBuzz	A prime example of overengineering something that can be done in a way more simple manner.	Go through the codebase, and ask _"Why would you want to do this? And why is it overkill here?"
The bowling game kata	A programming kata by uncle Bob. Apart from learning how he thinks, the exercise also focusses on suppressing your personal need to overly beautify a simple project.	Do the exercise and stop yourself from creating too many classes. Repeat the mantra: *"This is fine for now"* to suppress your urges to add indirection or OO concepts to the design.

Do not pass along too many parameters

Context

Functions (or methods) that take in a many input variables, also called parameters, are hard to read. The meaning and intent op each parameter is not always clear. Oftentimes, an abundance of input parameters is a sign of mixed responsibilities. Other times it indicates multiple versions of the same functionality, or it might indicate missing concepts in your model.

Drivers

- Method functionality should be visible at a glance
- It is initially easier to write one big function that can deal with a variety of inputs
- Maintenance becomes easier as code complexity decreases
- Testing becomes easier ot less inputs are needed (lees permutations == less risk of outlier flows)

Solution

- Consider the possibility that you placed the method in the wrong place. Moving it entirely to another class could reduce the need of the parameters being passed to it.
- Apply the *builder* or *factory* pattern ipf the abundance of parameters occurs in a constructor method
- Extract class fields or constants if able
- Split the method into multiple, smaller, methods

> **Author's note:** I must admit that most modern IDE's offer the programmer some help by displaying the names of the input parameters in the calling method's definition. This works great for your own code, but does not play very nice with external libraries. Even so, it is a good practice to keep your method definitions short, sweet, and to the point.

Examples

Let's revisit the code we looked at earlier. This time it's a version that does not comply with the unspoken rule of minimizing the amount of parameters.

```
private int score(int score, int rollCounter, int amountOfFramesInGame,
int maxPinsPerFrame) {
    for(int frame = 0; frame < amountOfFramesInGame; frame++) {
        if(isStrike(rollCounter, maxPinsPerFrame)) {
            score += maxPinsPerFrame + strikeBonus(rollCounter);
            rollCounter += 1;
        } else if(isSpare(rollCounter)) {
            score += sumOfPinsKnockedOverInFrame(rollCounter) +
spareBonus(rollCounter);
            rollCounter += 2;
        } else {
            score += sumOfPinsKnockedOverInFrame(rollCounter);
            rollCounter += 2;
        }
    }
    return score;
}
```

You might say this the four parameters do not look all too bad. At least the parameters have a clear name, that allows anyone that calls our code to know what we expect. While this is true when you look at the receiving method, take a look at how it looks from the caller's point of view:

```
public int main() {
    return score(0, 0, 10, 10);
}
```

If you are wondering how to solve this issue: there are a few possibilities. In the above example, most of the parameters we are sending into the method are fixed in nature. That means we can easily extract them to constants. On other occasions, you might notice that your are passing a value all the way down a method hierarchy within the same class. *Extracting a parameter to a class field* is another way you can fix this smell.

Sometimes it is hard to know how many parameters are okay, and when you are crossing the line. Some advocate that the maximum amount of parameters is two. Others adhere to a 4-parameters standard. It is almost impossible to just put a number on it. In general: the fewer parameters you use, the better.

> Try and use as few parameters as you can

Tests as a safety net

Property	Value
Submitter	Stijn Dejongh
Date of submission	15 Oct 2022
Type	`Practice`
Tags	`code testing`, `TDD`, `maintainability`, `speed`

Context

> [!NOTE] Add a textual description of the context in which this pattern can be applied

Drivers

> [!NOTE] A bullet list explaining why this pattern makes sense, try and be as objective as possible here

Enablers

> [!NOTE] Contextual factors that increase the chances of successfully implementing the pattern

Deterrents

> [!NOTE] Contextual factors that decrease the chances of successfully implementing the pattern

Solution

> [!NOTE] Describe the core idea of the pattern and how to apply it. Add subsections as you see fit in order to clearly communicate the idea

Measuring success

> [!NOTE] How do we know if we applied the pattern succesfully? What are our *'red flags'* that should trigger an adaption of the style of application?

Examples

Use Cases / Testimonials [Optional]

References

> Add links to other patterns or content in this collection (or external ones), please add references to the source material if you were inspired by someone else's work. Feel free to add your own previous work as a reference.

Item	Description
Some thingy	Why it is here

Resources

`/4_Software_development/Resources`

cli

`/4_Software_development/Resources/cli`

The command line interface, or CLI for short, is a powerful utility. Unfortunately, all this power can be quite overwhelming when you first become acquainted with it. Quite a few computer enthusiasts will remember the first time they accidentally found themselves in a `vi`-shell and having no idea how to get out of it.

Although intimidating, once you become familiar with using the command line, you might notice how it allows you to work much faster then using a GUI. The sections below contain some tips, tricks and configurations to level up your terminal.

> UNLIMITED POWER!
>
> ~ Emperor Palpatine

*NIX terminal tips

As developers, we are used to typing. Our brain is usually the limiting factor in using cli commands efficiently. Long commands, with multiple options are especially annoying. Luckily, most *NIX based systems have a good bash-like terminal out of the box. Let's take a look at making it work even better.

Bash aliases for developers

from gist bash_aliases

.bashrc

```
# TEXT aliases
alias v="vim"
alias nv="nvim"
alias em="emacs"

# GIT ALIASES

alias g=git
alias glog="git lg"
alias gsquash="git squash"

# PRODUCTIVITY ALIASES
alias inspire="fortune oblique"

# KEYBOARD ALIASES
alias workman="setxkbmap us; xmodmap ~/xmodmap/xmodmap.workman && xset r
66"
alias qwerty="setxkbmap us; xset -r 66"
alias colemak="setxkbmap us -variant colemak"

# MAVEN ALIASES

alias mci="mvn clean install"
alias mst="mci -Dmaven.test.skip=true"
alias mi="mvn install"
alias mrprep="mvn release:prepare"
alias mrperf="mvn release:perform"
alias mrrb="mvn release:rollback"
alias mdep="mvn dependency:tree"
alias mpom="mvn help:effective-pom"
alias msonar='mci -Psonar -Dsonar.host.url=http://localhost:9000 -
Dsonar.login="$SONAR_KEY"'

# DOCKER ALIASES
alias dockerps="docker ps --format 'table
{{.ID}}\t{{.Image}}\t{{.Status}}\t{{.Names}}'"

# Bash aliases
alias breload="source ~/.bashrc"
```

.gitconfig

```
# General git usage
squash = "!f(){ git reset --soft HEAD~${1} && git commit --edit -m\"$(git
log --format=%B --reverse HEAD..HEAD@{1})\"; };f"
lg = "log --color --graph --pretty=format:'%Cred%h%Creset -
%C(yellow)%d%Creset %s %Cgreen(%cr) %C(bold blue)<%an>%Creset' --abbrev-
commit"
c = "!git add -A && git commit -m "
```

```
wip = "!git add -A && git commit -m 'WIP commit - quick save'"
amno = "!git add -A && git commit --amend --no-edit"
amend = "!git add -A && git commit --amend"
work = !git lg --author \"Stijn\" --before today --after \"two days ago\"
--no-merges
info = !git st && echo && git lo && echo && git remote -v

# Handling files that are to be assumed unchanged.
ignore = update-index --assume-unchanged
uningore = update-index --no-assume-unchanged
ignored = !git ls-files -v | grep '^[a-z]'

# Branches
p = "!git push origin $(git rev-parse --abbrev-ref HEAD)"
rr = "!git fetch --all && git rebase origin/master"
n = "!git checkout -b"
co = "!git checkout"
st = "!git status"
rv = "!git checkout --"
cleanup = "!git reset --hard && git clean -f"

# Stashes
isl = "!git stash list"
sa = "!git stash apply"
ss = "!git stash save"

# Diffs
dt = difftool --no-prompt
mt = mergetool --no-prompt

# Tags
lstags = "!f(){ echo 'Describe: '; git describe; echo 'Latest 5 tags: ';
git tag | sort -V | tail -5; }; f"
colt = "!f(){ local latest=`git tag | sort -V | tail -1`; git checkout
$latest; }; f"

# Read aliases
alias = "config --get-regexp 'alias.*'"
```

Tutorials

/4_Software_development/Tutorials

As from time to time, we just want to be told how to do something: Here are some miniature courses, tutorials and small HOWTO articles related to Software Development.

Enjoy!

IntelliJ_Hotkeys

/4_Software_development/Tutorials/IntelliJ_Hotkeys

This tutorial series was originally published under the MIT License as `ProductivityWithShortcuts`, by its creator: Tim Schraepen.

> The MIT License (MIT)
>
> Copyright (c) 2016 Tim Schraepen
>
> Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:
>
> The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.
>
> THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

Tutorial structure

Each chapter contains a thorough explanation, and a set of exercises. Even though the progression in these chapters might seem to be increasing in difficulty, you don't need to think of them as thresholds. The main reason this series is divided into chapters is to help your brain make new neural pathways even better by trying to categorize certain types of shortcuts. This way, we hope you will think of Using them more frequently in real life.

> ![TIP] To speed up your tutorial progression, it is strongly advised to copy the entire source code used in the tutorial to your local machine. You can either fetch this code from the original authors GitHub page, or from our copied version.

Quality of life improvements

Besides a few visual upgrades over the original version of this series, this tutorial includes shortcuts for both the Windows and Macintosh operating systems. As they differ significantly in some places, different descriptions are given for each OS. In the online version of this text, you will find tabular panes like the one below.

Windows

Contains a description for the Windows operating system family.

Mac

Contains a description for the macOS operating system family.

What you will learn

- Increase your productivity by using IntelliJ shortcuts.
- In what situations/contexts shortcuts are helpful.
- Some patterns for multi-cursor usage.

How you will learn

By doing! The only way to get shortcuts into muscle memory is by using them. **All. Of. The. Darn. Time.** This tutorial series provides a structured approach to learn productivity shortcuts Using incrementally challenging exercises that will help you move forward at a consistent pace.

> [!TIP] To use a keyboard shortcut, press and hold one or more modifier keys and then press the last key of the shortcut.
> If you are using a keyboard made for a Windows PC while working on a Mac, use the `Alt` key instead of ⌥, and the Windows logo key (⊞) instead of ⌘.

Good Luck, Have Fun!

If you like these exercises, share them with your friends and colleagues, and drop a line to thank Timfor making this tutorial publicly available.

Code snippets

Package: be.swsb.productivity.common

Class `AFugly`

```
package be.swsb.productivity.common;

public class AFugly {
    private String eff;
    private String yew;
    private String gee;
    private String el;
    private String why;
    private Face face;

    AFugly(String eff, String yew, String gee, String el, String why,
Face face) {
        this.eff = eff;
        this.yew = yew;
        this.gee = gee;
        this.el = el;
        this.why = why;
        this.face = face;
    }

    public String getEff() {
        return eff;
```

```
    }

    public String getYew() {
        return yew;
    }

    public String getGee() {
        return gee;
    }

    public String getEl() {
        return el;
    }

    public String getWhy() {
        return why;
    }

    public Face getFace() {
        return face;
    }
}
```

Class `FuglyTestBuilder`

```
package be.swsb.productivity.common;

public class FuglyTestBuilder {

    private String eff;
    private String yew;
    private String gee;
    private String el;
    private String why;
    private Face face;

    public static FuglyTestBuilder fugly() {
        return new FuglyTestBuilder();
    }

    private FuglyTestBuilder() {}

    public AFugly build() {
        return new AFugly(eff, yew, gee, el, why, face);
    }

    public FuglyTestBuilder withEff(String eff) {
        this.eff = eff;
        return this;
    }
```

```
    public FuglyTestBuilder withYew(String yew) {
        this.yew = yew;
        return this;
    }

    public FuglyTestBuilder withGee(String gee) {
        this.gee = gee;
        return this;
    }

    public FuglyTestBuilder withEll(String el) {
        this.el = el;
        return this;
    }

    public FuglyTestBuilder withWhy(String why) {
        this.why = why;
        return this;
    }

    public FuglyTestBuilder withFace(Face face) {
        this.face = face;
        return this;
    }
}
```

Class `Face`

```
package be.swsb.productivity.common;

public class Face {
    private int eyes;
    private String color;
    private int nosewidth;

    public Face(int eyes, String color, int nosewidth) {
        this.eyes = eyes;
        this.color = color;
        this.nosewidth = nosewidth;
    }

    public int getEyes() {
        return eyes;
    }

    public String getColor() {
        return color;
    }

    public int getNosewidth() {
        return nosewidth;
```

```
    }
}
```

Class `FaceTestBuilder`

```
package be.swsb.productivity.common;

public class FaceTestBuilder {
    private int eyes;
    private String color;
    private int nosewidth;

    public static FaceTestBuilder face() {
        return new FaceTestBuilder();
    }

    private FaceTestBuilder() {}

    public Face build() {
        return new Face(eyes, color, nosewidth);
    }

    public FaceTestBuilder withEyes(int eyes) {
        this.eyes = eyes;
        return this;
    }

    public FaceTestBuilder withColor(String color) {
        this.color = color;
        return this;
    }

    public FaceTestBuilder withNosewidth(int nosewidth) {
        this.nosewidth = nosewidth;
        return this;
    }
}
```

Chapter 1 - Basics

Copy and Pasting

Windows

Using `ctrlC` and `ctrlV`, copy the Chapter1 constructor to create a new one without the number parameter, initialize the `number` field to the default value of `0`.

Use your mouse to select text, then press `ctrlC` to copy the selected text. Move your cursor to a blank line, then press `ctrlV` to paste the selected text.

Mac

Using `⌘C` and `⌘V`, copy the Chapter1 constructor to create a new one without the number parameter, initialize the `number` field to the default value of `0`.
Use your mouse to select text, then press `⌘C` to copy the selected text. Move your cursor to a blank line, then press, then press `⌘V` to paste the selected text.

```
public class Chapter1 {

    private String name;
    private int number;

    public Chapter1(String name, int number) {
        this.name = name;
        this.number = number;
    }

    // Exercise 1
    // Copy the Chapter1 constructor to create a new one without a
number, have the default number be 0.

    // Exercise 2
    // Do the same but with a default name of "Chapter". This time only
use your keyboard.

    // Exercise 3
    // Try to repeat Exercise 1, but instead of ctrl+c and ctrl+v, use
ctrl+shift+a to look up your copy and paste actions
}
```

> [!ATTENTION] Repeat the exercise, this time using **ONLY** your keyboard to select text. Turn your mouse upside-down if you have to!

How to look up any actions' shortcut

Windows

Repeat the previous exercise, but instead of using `ctrlC` / `ctrlV`, use `ctrlshifta` to look up the copy and paste actions in the quick help menu. Alternatively, you can double-tab the `shift` key do open the quick action menu.

Mac

Repeat the previous exercise, but instead of using `⌘C` / `⌘V`, use `⌘a` to look up the copy and paste actions in the quick help menu. Alternatively, you can double-tab the `shift` key do open the quick action menu.

IntelliJ's *Productivity Guide*

Windows

Open IntelliJ's Productivity Guide using these key combinations: First press `alth` for *(H)elp* (in the taskbar), then press `P` to select *(P)roductivity Guide*.

bonus: Try opening the Productivity Guide using `ctrlshifta`.

Mac

Open IntelliJ's Productivity Guide using these key combinations: First press `^F2` to focus on the taskbar, then use the arrow keys to navigate to the Help menu, then press down to expand the menu itself, then press `enter` to select *My Productivity*.

bonus: Try opening the Productivity Guide using `⌘shifta`.

Indenting code

Class `Fugly.java`

```
package be.swsb.productivity.chapter1.indentation;

import static be.swsb.productivity.common.FaceTestBuilder.face;
import static be.swsb.productivity.common.FuglyTestBuilder.fugly;

public class Fugly {

    public static void indentMeProperlyPlease() {

System.out.println(fugly().withEff("f").withYew("u").withGee("g").withEll
("l").withYew("y").withFace(face().withEyes(1).withColor("poop-
brown").withNosewidth(500).build()).build().toString());
    }

    public static void indentedItShouldLookLikeThis() {
        System.out.println(fugly()
                .withEff("f")
                .withYew("u")
                .withGee("g")
                .withEll("l")
                .withYew("y")
                .withFace(face()
                        .withEyes(1)
                        .withColor("poop-brown")
                        .withNosewidth(500)
                        .build())
                .build()
                .toString());
    }
```

```
}
```

Class `FuglyToo.java`

```
package be.swsb.productivity.chapter1.indentation;

import static be.swsb.productivity.common.FaceTestBuilder.face;
import static be.swsb.productivity.common.FuglyTestBuilder.fugly;

public class FuglyToo {

  public static void indentMeProperlyPlease() {
          System.out.println(fugly()
                  .withEff("f")
                                                  .withYew("u")
      .withGee("g")
                      .withEll("l")
                                                              .withYew("y")
              .withFace(face()
                                      .withEyes(1)
                          .withColor("poop-brown")
          .withNosewidth(500)
.build())
                                  .build()

.toString());
    }
}
```

Windows

Open Fugly.java, use selection and indent the test builder patterns properly. For this exercise, you can use `shift` and your arrow keys to select lines. Use `Tab` to indent them manually, or use `ctrlaltl` to automatically format the selected lines.

Hint: When manually indenting, first use `shiftTab` to unindent everything until the entire selection is against the left side, then `Tab` the entire selection into its first indentation, decrease your selection and `Tab` that into its second indentation. Rinse and repeat.

Mac

Open Fugly.java, use selection and indent the test builder patterns properly. For this exercise, you can use `shift` and your arrow keys to select lines. Use ⇥ to indent them manually, or use `^⌥l` to automatically format the selected lines.

Hint: When manually indenting, first use `shift`⇥ to unindent everything until the entire selection is against the left side, then ⇥ the entire selection into its first indentation, decrease your selection and ⇥ that into its

second indentation. Rinse and repeat.

Undo, Redo

Windows

In most editors, Redo is mapped to `ctrly`. Not so in IntelliJ.
This can lead to hilarious (or super annoying) situations where you'll lose your *undo buffer*.

Let's try it out and see what happens so you'll remember it better. Open `FuglyToo.java` once more. Add a comment above the method that reads `// this method is fugly`. Add a comment on a new line that reads `// such fugliness should never be allowed`. Press `ctrlz` (*Undo*) and see what happens.

Press it a couple times.

Now press `ctrlshiftz` (*Redo*) and see what happens. Add these three comments to the file, each starting on a different line:

```
// herpty
// derpty
// derp
```

After you've typed the last line, press `ctrlz` until you only have `// herpty` left.

As most people will have the reflex to press `ctrly` to *Redo* their work, let's see what happens when we do just that. Use `ctrly`, then try `ctrlshiftz` to attempt and redo the revert you wish to reapply.

Try `ctrlz` and see what that does. Try `ctrlshiftz` again now.

Keep this strange behavior in mind when you work in IntelliJ, or in another editor that doesn't have `ctrly` for *Redo* :)

Mac

The default undo and redo keyboard shortcuts on MacOS work the same over most applications. The weird behavior described in the windows section is not relevant for Mac users. Use `⌘z` to undo, and `⌘shiftz` to redo.

Congratulations! You finished the first chapter of the tutorial. If you learned a few new tricks, feel free to take a break and let the information sink in. Otherwise, we look forward to seeing you in chapter 2.

Chapter 2.1 - Navigation

Source code: Chapter Two

Chapter2.java

```
package be.swsb.productivity.chapter2;

public class Chapter2 {

    public Chapter2() {
        String phrase = theQuickBrownFoxJumpedOverTheLazyCamel();
    }

    private String theQuickBrownFoxJumpedOverTheLazyCamel() {
        return "The quick brown fox jumped over the lazy camel";
    }

    private String fox() {
        return "The quick brown fox ";
    }

    private String jumpUsingStrategy() { return
JumperStrategyFactory.epicJumperStrategy().jump(); }

    private String camel() {
        return "the lazy camel";
    }
}
```

EpicJumper.java

```
package be.swsb.productivity.chapter2;

public class EpicJumper implements Jumper {

    @Override
    public String jump() {
        return "pump epically ";
    }
}
```

Jumper.java

```
package be.swsb.productivity.chapter2;

public interface Jumper {

    String jump();
}
```

JumperStrategyFactory.java

```
package be.swsb.productivity.chapter2;

public class JumperStrategyFactory {

    public static Jumper epicJumperStrategy() {
        return new EpicJumper();
    }

    public static Jumper mehJumperStrategy() {
        return new MehJumper();
    }
}
```

MehJumper.java

```
package be.swsb.productivity.chapter2;

public class MehJumper implements Jumper {
    @Override
    public String jump() {
        return "pump ";
    }
}
```

Source code: Chapter Two/mud

BallRepository.java

```
package be.swsb.productivity.chapter2.mud.domain;

public class BallRepository {

    public Ball lookupById(String id) {
//        return new Ball(id, 10);
        return null;
    }
}
```

Ball.java

```java
package be.swsb.productivity.chapter2.mud.domain;

public class Ball {
    private final String id;
    private final int size;

    public Ball(String id, int size) {
        this.id = id;
        this.size = size;
    }

    public String getId() {
        return id;
    }

    public int getSize() {
        return size;
    }

    public void bounce(){
        System.out.println("bounce");
    }

    public void smash(){
        System.out.println("smash");
    }

    public void hit(){
        System.out.println("hit");
    }

    public void dribble(){
        System.out.println("dribble");
    }

    public void kick(){
        System.out.println("kick");
    }

    public void shoot(){
        System.out.println("shoot");
    }

    public void throwww(){
        System.out.println("throwww");
    }

    public void squeeze(){
        System.out.println("squeeze");
    }

    public void roll(){
```

```
        System.out.println("roll");
    }

    public void destroy(){
        System.out.println("destroy");
    }

    public void collect(){
        System.out.println("collect");
    }

    public void launch(){
        System.out.println("launch");
    }

    public void drizzle(){
        System.out.println("drizzle");
    }

    public void hoop(){
        System.out.println("hoop");
    }

    public void net(){
        System.out.println("net");
    }

    public void score(){
        System.out.println("score");
    }

    public void supersmash(){
        System.out.println("supersmash");
    }

    public void assemble(){
        System.out.println("assemble");
    }

    public void complete(){
        System.out.println("complete");
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;

        Ball ball = (Ball) o;

        if (size != ball.size) return false;
        return id != null ? id.equals(ball.id) : ball.id == null;

    }
```

```
    @Override
    public int hashCode() {
        int result = id != null ? id.hashCode() : 0;
        result = 31 * result + size;
        return result;
    }
}
```

RealBall.java

```
package be.swsb.productivity.chapter2.mud.domain;

public class RealBall extends Ball {

    public RealBall(String id, int size) {
        super(id, size);
    }

    public void bounce(){
        System.out.println("bounce");
    }

    public void smash(){
        System.out.println("smash");
    }

    public void hit(){
        System.out.println("hit");
    }

    public void dribble(){
        System.out.println("dribble");
    }

    public void kick(){
        System.out.println("kick");
    }

    public void shoot(){
        System.out.println("shoot");
    }

    public void throwww(){
        System.out.println("throwww");
    }

    public void squeeze(){
        System.out.println("squeeze");
    }
```

```
    public void roll(){
        System.out.println("roll");
    }

    public void destroy(){
        System.out.println("destroy");
    }

    public void collect(){
        System.out.println("collect");
    }

    public void launch(){
        System.out.println("launch");
    }

    public void drizzle(){
        System.out.println("drizzle");
    }

    public void hoop(){
        System.out.println("hoop");
    }

    public void net(){
        System.out.println("net");
    }

    public void score(){
        System.out.println("score");
    }

    public void supersmash(){
        System.out.println("supersmash");
    }

    public void assemble(){
        System.out.println("assemble");
    }

    public void complete(){
        System.out.println("complete");
    }

}
```

Word-skipping with the arrow keys

Windows

Open `Chapter2.java`, and place your cursor in front of the word *"The"* of the popular phrase, and tap `ctrl→` 8 times. You should now have your cursor at the beginning of the word *"camel"*.

Mac

Open `Chapter2.java`, and place your cursor in front of the *"The"* of the popular phrase, and tap ⌥→ 8 times. You should now have your cursor at the beginning of the word *"camel"*.

Jump to Start/End of Line

Windows

Still in `Chapter2.java`, place your cursor at the start of the popular phrase, and press `end`. Press `home` once, look at your cursors position, then press `home` again and see what happens. Press `home` one more time.

If you would like to see a stroboscopic effect, put your cursor at the beginning of the word *"return"*, and hold down `shifthome`.
Enjoy annoying anyone that might be watching over your shoulder.

Mac

Still in `Chapter2.java`, place your cursor at the start of the popular phrase, and press ⌘→. Press ⌘← once, look at your cursors position, then press ⌘← again and see what happens. Press ⌘← one more time.

If you would like to see a stroboscopic effect, put your cursor at the beginning of the word *"return"*, and hold down `shift⌘←`.
Enjoy annoying anyone that might be watching over your shoulder.

Jump to Begin/End of File

Windows

Try out `ctrlhome` and `ctrlend` in `Chapter2.java`.
Alternatively, you can accomplish the same using `PgUp` and `PgDn`.

Mac

Try out `Fn⌘←` and `Fn⌘→` in `Chapter2.java`.

CamelHumps (+ how to toggle)

Windows

In `Chapter2.java`, put your cursor at the beginning of the method `theQuickBrownFoxJumpedOverTheLazyCamel`.

Try to use *Skip Word* with `ctrl→` on that method.

Depending on your CamelHumps setting, your cursor either ended up on the *"Q"* or it skipped the entire method name and ended up on the *"("*.

Return to the beginning of the method name and press `ctrlshifta`, then type `CamelHump`. There used to be a setting named `Smart Keys: Use "CamelHumps" words` with a toggle indicator, but this

disappeared since some new release around 2019.

Instead if you want to do something using the alternative CamelHumps mode, there **is** an action in the action menu for that.

These are called `Move Caret to Next Word with Selection in Different "CamelHumps" Mode` (or Previous Word, or without Selection).

They all appear when you type CamelHumps and IntelliJ remembers your last action command, so it's not all bad.

> [!TIP] Other tools, like SublimeText, have different key combinations to skip the entire word (`ctrl→`), or skip based on CamelCasing (`alt→`).

Mac

> [!NOTE] These shortcuts might override with your OS shortcuts, so it's wise to disable these before you continue with the next exercises.

In `Chapter2.java`, put your cursor at the beginning of the method `theQuickBrownFoxJumpedOverTheLazyCamel`.

Try to use *Skip Word* with ⌥→ on that method.

Depending on your CamelHumps setting, your cursor either ended up on the *"Q"* or it skipped the entire method name and ended up on the *"("*.

Return to the beginning of the method name and press⌘a, then type `CamelHump`. There used to be a setting named `Smart Keys: Use "CamelHumps" words` with a toggle indicator, but this disappeared since some new release around 2019.

Instead if you want to do something using the alternative CamelHumps mode, there **is** an action in the action menu for that.

These are called `Move Caret to Next Word with Selection in Different "CamelHumps" Mode` (or Previous Word, or without Selection).

They all appear when you type CamelHumps and IntelliJ remembers your last action command, so it's not all bad.

> [!TIP] Other tools, like SublimeText, have different key combinations to skip the entire word (⌃→), or skip based on CamelCasing (⌥→).

Jumping methods

Windows

In `Chapter2.java`, place your cursor at the `theQuickBrownFoxJumpedOverTheLazyCamel` method.
Press `alt↓` a few times and see what happens.
Now use `alt↑` to go back the way you came.

Mac

In `Chapter2.java`, place your cursor at the `theQuickBrownFoxJumpedOverTheLazyCamel` method.
Press `^Shift↓` a few times and see what happens.
Now use `^Shift↑` to go back the way you came.

Jump to "error"

Windows

Move to the top of the file with `ctrlHome` and from there press `F2`. This should navigate your cursor to the class named `Chapter2` because IntelliJ marks it as being *unused*.

If you keep pressing `F2` it should keep cycling your cursor over the *unused* warnings. In between the methods `jump()` and `camel()`, paste the following:

```
privet String kakdilla() {
    "horocho";
}
```

Move to the top of the file again, and press `F2` once again. Notice how the cursor now first jumps to the actual compilation error (*`privet` cannot be resolved*).

> [!NOTE] Cycling only happens over all actual *errors*, and the *unused warnings* are not cycled over anymore.

Mac

Move to the top of the file with `Fn⌘←` and from there press `F2`. This should navigate your cursor to the class named `Chapter2` because IntelliJ marks it as being *unused*.

If you keep pressing `F2` it should keep cycling your cursor over the *unused* warnings. In between the methods `jump()` and `camel()`, paste the following:

```
privet String kakdilla() {
    "horocho";
}
```

Move to the top of the file again, and press `F2` once again. Notice how the cursor now first jumps to the actual compilation error (*`privet` cannot be resolved*).

> [!NOTE] Cycling only happens over all actual *errors*, and the *unused warnings* are not cycled over anymore.

Jump into

Windows

"Jump into", or **"drill down"** as I like to call it, allows you to follow the path the code will execute at runtime. It is a big timesaver when attempting to follow the logic of any given program.

Right now, there's a typo in both the `EpicJumper.java` and `MehJumper.java` classes. Let's fix that.

Go to `Chapter2.jumpUsingStrategy()` and place your cursor on the `jump()` method call. Press `ctrl b`. This should take you straight to the interfaces `jump()` method.

Now, let's go back to where we came from. Press `ctrl alt b`. IntelliJ knows you want to **"drill down"** into the actual method implementation but doesn't know which one, so it will suggest some options. Select the `MehJumper` method by pressing `↓` and then `enter` and see where it leads you.

You can now correct the typo in the method, and move on to the next exercise.

Mac

"Jump into", or **"drill down"** as I like to call it, allows you to follow the path the code will execute at runtime. It is a big timesaver when attempting to follow the logic of any given program.

Right now, there's a typo in both the `EpicJumper.java` and `MehJumper.java` classes. Let's fix that.

Go to `Chapter2.jumpUsingStrategy()` and place your cursor on the `jump()` method call. Press `⌘ b`. This should take you straight to the interfaces `jump()` method.

Now, let's go back to where we came from. Press `⌘ ⌥ b`. IntelliJ knows you want to **"drill down"** into the actual method implementation but doesn't know which one, so it will suggest some options. Select the `MehJumper` method by pressing `↓` and then `enter` and see where it leads you.

You can now correct the typo in the method, and move on to the next exercise.

> [!NOTE] As this chapter is quite lengthy, we split it up into two parts. Feel free to take a break now, or continue on to Chapter 2.2.

Chapter 2.2 - Navigation (Continuation)

> [!NOTE] As this chapter is quite lengthy, we split it up into two parts. Make sure you have completed Chapter 2.1 before continuing.

History and its importance

Windows

In the previous exercise we drilled down into a method call and changed some things. But sometimes we want to go back in time (usually after messing something up). Let's repeat the previous exercise! If you are following this tutorial in one go, and are currently at the end-position of the previous topic, you can continue onwards from there.

Press `ctrl alt ←` to go return to your starting position. You should now be back at the `Chapter2.java` class.

Now repeat the previous exercise, but pick the `EpicJumper` and also fix the typo. Then go back again using `ctrl alt ←`.

Also try backtracking your backtrack by pressing `ctrl alt →`.

> [!NOTE] every time you use **Navigation shortcuts** that bring you to new classes, IntelliJ will remember this in a Navigation History of sorts.

Mac

In the previous exercise we drilled down into a method call and changed some things. But sometimes we want to go back in time (usually after messing something up). Let's repeat the previous exercise! If you are following this tutorial in one go, and are currently at the end-position of the previous topic, you can continue onwards from there.

Press ⌘ ⌥ ← to go return to your starting position. You should now be back at the `Chapter2.java` class.

Now repeat the previous exercise, but pick the `EpicJumper` and also fix the typo. Then go back again using ⌘ ⌥ ←.

Also try backtracking your backtrack by pressing ⌘ ⌥ →.

> [!NOTE] every time you use **Navigation shortcuts** that bring you to new classes, IntelliJ will remember this in a Navigation History of sorts.

Jump to last edit position

Windows

From the end of previous exercise, make sure you're back in the `Chapter2.java` class and press `ctrl shift backspace` to return to where you were last editing. Try pressing the hotkey again and see what happens.

Mac

From the end of previous exercise, make sure you're back in the `Chapter2.java` class and press ⌘ `shift backspace` to return to where you were last editing. Try pressing the hotkey again and see what happens.

Show in Project Window

Windows

Open `MehJumper.java` by pressing `ctrl n`, then use `alt F1` to open up the `Project` navigational sidebar with the `MehJumper.java` class selected.

You can now use `alt 1` (**do not press F1, we mean the actual digit**) to minimize the sidebar and move your window focus back to your editor.

Mac

Open `MehJumper.java` by pressing ⌘ `o`, then use ⌥ `F1`, `enter` to open up the `Project` navigational sidebar with the `MehJumper.java` class selected.

You can now use ⌥ `1` (**do not press F1, we mean the actual digit**) to minimize the sidebar and move your window focus back to your editor.

More navigational goodness: code hierarchy transversal

Windows

Inspect the `mud` package. It's got your typical layered application where we pass around a `Ball` through all of its layers. In order to navigate more complex code hierarchies, play around with some of these hotkeys:

Pressing `alt F7` will show you how the selected element is used inside your codebase. This shortcut works on virtually anything, be it a class, a method or a field in a separate **Tool Window**. As an example: open `Ball.java` using `ctrl n`, and press your arrow keys to navigate to the`getId()` method inside this class. Now press `alt F7` and look at the bottom of your screen.

`ctrl alt h` will show you the call hierarchy leading up to the element you are currently inspecting. Repeat the previous step, but instead of inspecting an element's usage, press `ctrl alt h`. Navigate the element tree using the arrow keys, and select any element you wish to take a closer look at with `ctrl enter` (or `F4` if you want to navigate to the code without further ado).

> [!TIP] move this navigational **Tool Window** to the bottom bar (next to `3: Find`), because you'll usually want to optimize your screen's horizontal space rather than its vertical space.

Mac

Inspect the `mud` package. It's got your typical layered application where we pass around a `Ball` through all of its layers. In order to navigate more complex code hierarchies, play around with some of these hotkeys:

Pressing ⌥ `F7` will show you how the selected element is used inside your codebase. This shortcut works on virtually anything, be it a class, a method or a field in a separate **Tool Window**. As an example: open `Ball.java` using ⌘ `o`, and press your arrow keys to navigate to the`getId()` method inside this class. Now press ⌥ `F7` and look at the bottom of your screen.

⌃ ⌥ `h` will show you the call hierarchy leading up to the element you are currently inspecting. Repeat the previous step, but instead of inspecting an element's usage, press ⌃ ⌥ `h`. Navigate the element tree using the arrow keys, and select any element you wish to take a closer look at with `ctrl enter` (or `F4` if you want to navigate to the code without further ado).

> [!TIP] move this navigational **Tool Window** to the bottom bar (next to `3: Find`), because you'll usually want to optimize your screen's horizontal space rather than its vertical space.

Jump to definition

Windows

Navigate to the `return id;` line in the `Ball.java` class and put your cursor on `id`. Now press `ctrl b` This should navigate your code editor to the instantiation of the field itself. Press `ctrl b` again. This time it should

show a popup asking if you want to show **accessors** of the field. Let's go with `Yes`.

Mac

Navigate to the `return id;` line in the `Ball.java` class and put your cursor on `id`. Now press `⌥ b` This should navigate your code editor to the instantiation of the field itself. Press `⌥ b` again. This time it should show a popup asking if you want to show **accessors** of the field. Let's go with `Yes`.

Pop-up windows (but not the annoying kind)

Windows

Sometimes when you are working on code, you want to quickly reference how a certain class, field or method is defined without opening a new workspace window. In order to do so, you can make use of the `ctrl shift i` keyboard combination to do just so.

Other useful overlay pop-ups include: the **quick documentation** and **quick parameter definition** shortcuts. Let's find out what they do! Move to any line of code, and press `ctrl q`. The overlayed information pop-up will show you relevant documentation of the selected code element.

Navigate to the `return id;` line in the `Ball.java` class and put your cursor on `id` again. Pressing `ctrl p` will show you the relevant documentation for this parameter. As we have not written any documentation, this overlay window will be blank.

Mac

Sometimes when you are working on code, you want to quickly reference how a certain class, field or method is defined without opening a new workspace window. In order to do so, you can make use of the `⌥ space` keyboard combination to do just so.

Other useful overlay pop-ups include: the **quick documentation** and **quick parameter definition** shortcuts. Let's find out what they do! Move to any line of code, and press `^ j`. The overlayed information pop-up will show you relevant documentation of the selected code element.

Navigate to the `return id;` line in the `Ball.java` class and put your cursor on `id` again. Pressing `⌘ p` will show you the relevant documentation for this parameter. As we have not written any documentation, this overlay window will be blank.

More source code!

The next sections will refer to even more source code. For completeness, we will provide it here:

BallService.java

```
package be.swsb.productivity.chapter2.mud.service;

public interface BallService {
    BallTO findBall(String id);
}
```

BallServiceImpl.java

```
package be.swsb.productivity.chapter2.mud.service;

import be.swsb.productivity.chapter2.mud.domain.Ball;
import be.swsb.productivity.chapter2.mud.domain.BallRepository;

public class BallServiceImpl implements BallService {

    private BallRepository ballRepository;
    private BallTOAssembler ballTOAssembler;

    public BallServiceImpl(BallRepository ballRepository) {
        this.ballRepository = ballRepository;
    }

    @Override
    public BallTO findBall(String id) {
        Ball ball = ballRepository.lookupById(id);
        return ballTOAssembler.assembleTOFrom(ball);
    }
}
```

BallTO.java

```
package be.swsb.productivity.chapter2.mud.service;

public class BallTO {
    private String id;
    private int size;

    public BallTO() {
    }

    public String getId() {
        return id;
    }

    public void setId(String id) {
        this.id = id;
    }

    public int getSize() {
        return size;
    }

    public void setSize(int size) {
```

```
        this.size = size;
    }
}
```

BallTOAssembler.java

```
package be.swsb.productivity.chapter2.mud.service;

import be.swsb.productivity.chapter2.mud.domain.Ball;

public class BallTOAssembler {
    public BallTO assembleTOFrom(Ball ball) {
        BallTO ballTO = new BallTO();
        ballTO.setId(ball.getId());
        ballTO.setSize(ball.getSize());

        return ballTO;
    }
}
```

Line-based navigation

Here is a part of a very long stacktrace.

```
    java.lang.NullPointerException
      at
be.swsb.productivity.chapter2.mud.service.BallServiceImpl.findBall(BallSe
rviceImpl.java:18)
      at
be.swsb.productivity.chapter2.mud.ui.BallScreen.render(BallScreen.java:15
)
      at
be.swsb.productivity.chapter2.mud.ui.BallScreenTest.screenCallsStuff(Ball
ScreenTest.java:11)
      at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
      at
sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java
:62)
      at
sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorI
mpl.java:43)
      at java.lang.reflect.Method.invoke(Method.java:497)
      at
org.junit.runners.model.FrameworkMethod$1.runReflectiveCall(FrameworkMeth
od.java:47)
      at
org.junit.internal.runners.model.ReflectiveCallable.run(ReflectiveCallabl
```

```
e.java:12)
      at
org.junit.runners.model.FrameworkMethod.invokeExplosively(FrameworkMethod
.java:44)
      at
org.junit.internal.runners.statements.InvokeMethod.evaluate(InvokeMethod.
java:17)
      at org.junit.runners.ParentRunner.runLeaf(ParentRunner.java:271)
      at
org.junit.runners.BlockJUnit4ClassRunner.runChild(BlockJUnit4ClassRunner.
java:70)
      at
org.junit.runners.BlockJUnit4ClassRunner.runChild(BlockJUnit4ClassRunner.
java:50)
      at org.junit.runners.ParentRunner$3.run(ParentRunner.java:238)
      at org.junit.runners.ParentRunner$1.schedule(ParentRunner.java:63)
      at
org.junit.runners.ParentRunner.runChildren(ParentRunner.java:236)
      at org.junit.runners.ParentRunner.access$000(ParentRunner.java:53)
      at org.junit.runners.ParentRunner$2.evaluate(ParentRunner.java:229)
      at org.junit.runners.ParentRunner.run(ParentRunner.java:309)
      at org.junit.runner.JUnitCore.run(JUnitCore.java:160)
      at
com.intellij.junit4.JUnit4IdeaTestRunner.startRunnerWithArgs(JUnit4IdeaTe
stRunner.java:119)
      at
com.intellij.junit4.JUnit4IdeaTestRunner.startRunnerWithArgs(JUnit4IdeaTe
stRunner.java:42)
      at
com.intellij.rt.execution.junit.JUnitStarter.prepareStreamsAndStart(JUnit
Starter.java:234)
      at
com.intellij.rt.execution.junit.JUnitStarter.main(JUnitStarter.java:74)
      at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
      at
sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java
:62)
      at
sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorI
mpl.java:43)
      at java.lang.reflect.Method.invoke(Method.java:497)
      at
com.intellij.rt.execution.application.AppMain.main(AppMain.java:144)
```

Windows

Let's see where the NullPointer is occurring. Open `BallServiceImpl`, using `ctrl g`. Now use `ctrl g` and enter the desired line number (this is line nr. `18`), as the stacktrace above states the error occurs on this line:

```
at
be.swsb.productivity.chapter2.mud.service.BallServiceImpl.findBall(BallServiceIm
pl.java:18)
```

Mac

Let's see where the NullPointer is occurring. Open `BallServiceImpl`, using ⌘ `l`. Now use ⌘ `l` and enter the desired line number (this is line nr. `18`), as the stacktrace above states the error occurs on this line: `at be.swsb.productivity.chapter2.mud.service.BallServiceImpl.findBall(BallServiceImpl.java:18)`

Navigate to method

Windows

Another way of navigating to the location of the code causing our NullPointer exception, albeit less precise, is to first copy the method name from the stacktrace, in this case `findBall`. Next, open `BallServiceImpl.java`, using `ctrl n`. Now press `ctrl F12`. This lists all methods of a class. To navigate to the suspicious method: paste the name we copied from the stacktrace into the method list popup window. You can then press `Enter` to navigate to that method.

Let's imagine we want to write a comparator for some object, or want to use it in a Set. You'll want to take a look at that objects `equals()` implementation. In case the object is filled to the brim with other methods, it might be easier to tap `ctrl F12` to check for an `equals` method. If you can't find it on your first try, tap `ctrl F12` again to browse the methods of the superclass as well. Let's open `RealBall.jave`, and try to find out if this class has its own `equals` and `hashcode` implementations, or wether makes use of its superclass implementations.

Mac

Another way of navigating to the location of the code causing our NullPointer exception, albeit less precise, is to first copy the method name from the stacktrace, in this case `findBall`. Next, open `BallServiceImpl.java`, using ⌘ `o` . Now press `ctrl F12`. This lists all methods of a class. To navigate to the suspicious method: paste the name we copied from the stacktrace into the method list popup window. You can then press `Enter` to navigate to that method.

Let's imagine we want to write a comparator for some object, or want to use it in a Set. You'll want to take a look at that objects `equals()` implementation. In case the object is filled to the brim with other methods, it might be easier to tap ⌘ `F12` to check for an `equals` method. If you can't find it on your first try, tap ⌘ `F12` again to browse the methods of the superclass as well. Let's open `RealBall.jave`, and try to find out if this class has its own `equals` and `hashcode` implementations, or wether makes use of its superclass implementations.

Chapter 3 - Selection

Source code: Chapter Three

Chapter2.java

```
package be.swsb.productivity.chapter3;
```

```
public class Chapter3 {

    public String selectStuff() {
        return "The quick brown fox " + "jumps over " + "the LazyDawg";
    }
}
```

Move + Select

Windows

Open `Chapter3.java`, then move your cursor to the beginning of the line containing `"jumps over"`, hold down `ctrl`, `shift` and press the right arrow key → to select that sentence. Now cut and paste it behind the first line.

Mac

Open `Chapter3.java`, then move your cursor to the beginning of the line containing `"jumps over"`, hold down ⌘, `shift` and press the right arrow key → to select that sentence. Now cut and paste it behind the first line.

Expand Selection

Windows

Press `ctrl n` and open `Fugly.java`. Move your cursor to line `23`, column `28` by pressing `ctrl g` and typing `23:28`. From that position press `ctrl w`, press this key combination again a few times, and see what happens.

Now try using `ctrl shift w` to incrementally reduce the size of your selection. Play around with both the `Expand selection` and `Reduce selection` key combinations until you feel comfortable with them.

Mac

Press ⌘ `o` and open `Fugly.java`. Move your cursor to line `23`, column `28` by pressing ⌘ `l` and typing `23:28`. From that position press ⌥ ↑, press this key combination again a few times, and see what happens.

Now try using ⌥ `shift` ↑ to incrementally reduce the size of your selection. Play around with both the `Expand selection` and `Reduce selection` key combinations until you feel comfortable with them.

Using Selection to help Navigation (e.g. Fluent API)

Windows

Because `ctrl w` expands a selection, and because the arrow keys decide where our cursor is going to be: ← at the beginning or → at the end of the selection.

We can use a little trick to format our Fluent API. Format the one-liner so that it looks like the method below. You can do this by pressing `ctrl w` until you have a selection containing a *"method call"*, e.g. `fugly()`, then

press → to put your cursor at the end, and press ⏎ (`enter`). Rinse and repeat.

Mac

Because ⌥ ↑ expands a selection, and because the arrow keys decide where our cursor is going to be: ← at the beginning or → at the end of the selection.

We can use a little trick to format our Fluent API. Format the one-liner so that it looks like the method below. You can do this by pressing ⌥ ↑ until you have a selection containing a *"method call"*, e.g. `fugly()`, then press → to put your cursor at the end, and press ⏎ (`enter`). Rinse and repeat.

Wrapping (IntelliJ feature)

Windows

IntelliJ has a neat feature that wraps your selection with braces, curly braces, single or double quotes, ... You can enable this feature under `Settings` by pressing ⌘ `>,`, then go to `Editor > General > Smart Keys`, and enable `Surround selection on typing quote or brace`.

An alternative is by pressing `ctrl shift a`, and then type `Smart Keys Braces`. Fix the `wrapStuff` method by selecting `"effffff"` with `ctrl w`, and then type a `"`. Then apply the same pattern to `yewwww` but instead of a `"`, type a `(`.Then apply the same pattern to the entire method body but type a `{`.

This feature really shines in combination with multi-cursor (which we'll see in a later chapter).

Mac

IntelliJ has a neat feature that wraps your selection with braces, curly braces, single or double quotes, ... You can enable this feature under `Settings` by pressing ⌘ `,`, then go to `Editor > General > Smart Keys`, and enable `Surround selection on typing quote or brace`.

An alternative is by pressing ⌘ `shift a`, and then type `Smart Keys Braces`. Fix the `wrapStuff` method by selecting `"effffff"` with ⌥ ↑s, and then type a `"`. Then apply the same pattern to `yewwww` but instead of a `"`, type a `(`.Then apply the same pattern to the entire method body but type a `{`.

This feature really shines in combination with multi-cursor (which we'll see in a later chapter).

Chapter 4 - Line Editing

> [!NOTE] Authors pro-tip: Listen to this song while working through the exercises

Source code: Chapter 4

```
package be.swsb.productivity.chapter4;

public class Chapter4 {

    /*
```

```
     * This is an unnecessary comment, so.... duplicate me :)
     * Delete all of these lines using ctrl+y
     */
    public Chapter4() {
    }

    private void _2_snarf() {
        System.out.println("Lion-O");
    }

    private void _1_lionO() {
        System.out.println("Snarf");
    }

    private void _3_cheetara() {
    }

    private String[] _4_thundercats() {
        return new String[]{ "lion-o", "jaga", "panthro", "tygra",
"cheetara", "snarf", "wily kit", "wily kat" };
    }
}
```

Duplicate line

Windows

Open `Chapter4.java`, move your cursor to line 6 (try and use a shortcut for this), and press `ctrl d`.

Mac

Open `Chapter4.java`, move your cursor to line 6 (try and use a shortcut for this), and press ⌘ `d`.

Yank

Windows

Press `ctrl y` repeatedly to delete the lines.

Mac

Press ⌘ `backspace` repeatedly to delete the lines.

Moving lines with and without constraints

Windows

While in `Chapter4.java`, the `System.out.println` function calls are switched around. Place your cursor on one of the `System.out.println` lines, hold down `alt shift` and press ↑ or ↓ to move that line. Do the same for the other line.

You will see the methods aren't in the desired order yet, so place your cursor on `_2_snarf`'s method signature. This time hold down `ctrl shift` and press ↑ or ↓ to move the entire method.

Mac

While in `Chapter4.java`, the `System.out.println` function calls are switched around. Place your cursor on one of the `System.out.println` lines, hold down ⌥ `shift` and press ↑ or ↓ to move that line. Do the same for the other line.

You will see the methods aren't in the desired order yet, so place your cursor on `_2_snarf`'s method signature. This time hold down `shift` ⌘ and press ↑ or ↓ to move the entire method.

Start new line

Windows

In `Chapter4.java`, jump to 20:20 (using `ctrl g`). From this position we want to start implementing the body of the method. Typically, one would do this by pressing `end`, and then `enter`. But you can do this in one go by pressing `shift enter`, so let's do just that.

This will come in handy later.

Mac

In `Chapter4.java`, jump to 20:20 (using ⌘ `l`). From this position we want to start implementing the body of the method. Typically, one would do this by pressing `end`, and then `enter`. But you can do this in one go by pressing `shift enter`, so let's do just that.

Join lines

Windows

We want to write the return statement of `_4_thundercats()` on just one line. Don't use a sequence of `end delete` combinations. Instead, first select all the thundercats' names (the strings), then press `ctrl shift j` (for Join lines).

> [!NOTE] Hint: use `ctrl w` right after the `{` character.

You might have to repeat the `ctrl shift j` combination, because IntelliJ's auto-formatting tends to kick in sometimes.

Mac

We want to write the return statement of `_4_thundercats()` on just one line. Don't use a sequence of ⌘ →, `delete` combinations. Instead, first select all the thundercats' names (the strings), then press ^ `shift j` (for Join lines).

> [!NOTE] Hint: use ⌥ ↑ right after the `{` character.

You might have to repeat the ^ `shift j` combination, because IntelliJ's auto-formatting tends to kick in sometimes.

Chapter 5 - Embedded Windows

IntelliJ has various *Tool Windows*, like the `1: Project` window, `9: Version Control` window, or `3: Find` window. When one of these *embedded* windows have focus, other shortcuts are available. In this topic we'll talk about some of them.

Opening/Closing (Toggling)

Windows

All of these windows are accessible by holding down `alt` and pressing the associating number. e.g. If you want to open or close the `1: Project` window, you press `alt 1`. You can also minimize the current active tool window using `shift escape`.

Mac

All of these windows are accessible by holding down ⌘ and pressing the associating number. e.g. If you want to open or close the `1: Project` window, you press ⌘ `1`.

Switching tabs

Windows

Some of these windows have multiple tabs in them, i.e. the `6: TODO` window. So let's open that with `alt 6`. You'll notice that it contains the *Project*, *Current File*, *Scope Based* and *Default Changelist* tabs. You can switch between these tabs by pressing `alt` → and `alt` ←.

Give it a try!

Mac

Some of these windows have multiple tabs in them, i.e. the `6: TODO` window. So let's open that with ⌘ `6`. You'll notice that it contains the *Project*, *Current File*, *Scope Based* and *Default Changelist* tabs. You can switch between these tabs by pressing ⌘ `shift [` and ⌘ `shift ]`.

Give it a try!

Navigation from Embedded Windows

Windows

Some Tool Windows will display results, like `3: Find` and `8: Hierarchy`, which you can use to navigate to directly.

There's two ways of doing this:

- `ctrl enter`: allows you to navigate to your selected result, but focus remains on your Tool Window
- `F4`: navigates to your selected result AND focuses the editor window in one go.

Let's try 'em both out.

We want to follow the path the code takes at runtime until we get to `CoffeeBeans.scent()`, so let's trace back our steps from there by opening `CoffeeBeans.java`, and `ctrl alt h` on the `scent()` method.

Use `ctrl enter` on `CoffeeSmeller.smell()`, see where it takes you, then `ctrl enter` on `Chapter5.smellBeans()`.

We figured out we want to change something in `CoffeeSmeller`, so select `CoffeeSmeller` in the `8: Hierarchy` tool window and instead of pressing `ctrl enter`, press `F4`.

Now we can change the implementation.

Mac

Some Tool Windows will display results, like `3: Find` and `8: Hierarchy`, which you can use to navigate to directly.

There's two ways of doing this:

- `⌥ enter`: allows you to navigate to your selected result, but focus remains on your Tool Window
- `⌘ ↑`: navigates to your selected result AND focuses the editor window in one go.

Let's try 'em both out.

Why resizing is for dummies

Windows

Stop resizing your *Tool Windows*, use them when you need them (see *Opening/Closing (Toggling)*).

If you don't need your *Tool Window*, you'll want to focus back on your editor window. You can do this from anywhere (meaning, from any focussed *Tool Window*) by pressing `ctrl shift F12`.

So, let's first imagine we were looking at a hierarchy of `CoffeeBeans.scent()`, we looked at `CoffeeSmeller`'s use and we checked where in the package structure `CoffeeSmeller` was situated by pressing `alt F1`.

Now we have both the `1: Project` and `8: Hierarchy` *Tool Windows* open. But we want to continue tweaking the CoffeeSmellers code, so let's press `ctrl shift F12` and get our focus back on where it belongs, without any distractions.

Happy editing!

Mac

Stop resizing your *Tool Windows*, use them when you need them (see *Opening/Closing (Toggling)*).

If you don't need your *Tool Window*, you'll want to focus back on your editor window. You can do this from anywhere (meaning, from any focussed *Tool Window*) by pressing `shift ⌘ F12`.

So, let's first imagine we were looking at a hierarchy of `CoffeeBeans.scent()`, we looked at `CoffeeSmeller`'s use and we checked where in the package structure `CoffeeSmeller` was situated by pressing `⌥ F1`.

Now we have both the `1: Project` and `8: Hierarchy` *Tool Windows* open. But we want to continue tweaking the CoffeeSmellers code, so let's press `shift ⌘ F12` and get our focus back on where it belongs, without any distractions.

Happy editing!

X_Appendix

`/X_Appendix`

Bibliography

- Grad, Burton. 2019. IEEE STARS article: Software Industry. digital publication on [https://ethw.org/Software_Industry](https://ethw. org/Software_Industry), retrieved on 29 october 2022.

Acknowledgements

Changelog

`/X_Appendix/Changelog`

Keeping a Changelog

All notable changes to this project will be documented in this file.

The format is based on Keep a Changelog, and this project adheres to Semantic Versioning.

Following changes are known:

- **Added** for new features.
- **Changed** for changes in existing functionality.
- **Deprecated** for soon-to-be removed features.
- **Removed** for now removed features.
- **Fixed** for any bug fixes.
- **Security** in case of vulnerabilities.

Changes

v1.1.0

Changed

- Update README to include references to tech Stack
- Update the script used to generate the documentation
- Update gitignore file for IntelliJ artifacts and configurations

Added

- Template: add pattern template to github issue tracker and update labels
- Productivity: Essay on agile software development

- Programming: Patterns on writing clean code
    - Baptize your code: the value of giving code artifacts proper names
- Productivity: Patterns on doing things and making progressing
    - Add Return on Investment pattern
    - Add OPERAS method pattern
- Add a Glossary containing terminology used in this tome
- Add LUA filter to include page breaks into the markdown
- GitOps: Quality of life gitops updates:
    - Add Issue template for pattern suggestions
    - Add Issue template for Learning Resource recommendations/Reviews
    - Add labels
    - Add contributors automation
- Automation:
    - Add JustDoIt script
    - Add script to create new section directory structure and empty files

Deleted

- Runner for docsite and docsite example output
- Old example documents
- DocSite as deployment of HTML files will not happen any time soon

v1.0.0 Initial Version

Added

- Repository structure
- Add documentation outlines
    - Add overview pages per sections
    - Add high-level description of software architecture

Current Working version

Changed

- formatted the patterns in `Software Development`
- extracted appendices `Glossary`, `Changelog`, and `Reading List` to separate folders
- main README has been updated
- apply formatting to Markdown documents
- moved "write good enough code" to "practices" from "concepts"
- sidebar now uses "folder" style instead of arrows
- update pattern template to follow the new heading structure
- OPERAs pattern now features collapsible sections and follows the intended pattern-practice structure
- Moved `Practices` and `Concepts` pattern categories to the top level of each pilar
- Update *"ROI"* pattern structure
- visual style of sidenav
- default pattern template has been updated

Added

- support for footnotes
- docsify.js notifications plugin
- styling to layout images
- included information about Pattern Languages and meta-modelling
- add description of knowledge sharing
- animated image of local docsify usage
- add `IntelliJ hotkeys` tutorial, by Tim Schraepen
    - Convert + Add chapter one
    - Convert + Add chapter two
- add "External Memory" Pattern
- add footnote plugin
- added syntax highlighting
- Section 0_Pattern_Language as it is not really relevant and should be added to the main README instead

Removed

- automatic glossary creation (issue with titles)
- automated changelog aggregation
- cleanup of unused/confusing plugins

Fixed

- images in software development patterns are now smaller
- spelling mistakes
- glossary is now a self-maintained section, without automated links

v2.0.0

Changed

- Restructured Content to fit the docsify structure
- Rearranged hierarchy to be domain-based over type-based
- Structure descriptions

Added

- Docsify deployment config
- New project banner image
- Glossary and Side navigation plugin
- Cover page and logo
- Reading list
- Sidebar directory markings (arrows)
- Patterns on Learning: ShuHaRi and Dreyfus
- Patterns on leadership: Lewin's leadership styles

Removed

- Dockerized document generation support
- Javascript slideshow library (for now)
- Unneeded duplicated descriptions

Fixed

- Resolved various typo's
- Automated build and deployment pipeline has been updated
- general quality upgrades (grammatical fixes, layout fixes, ...)

Glossary

`/X_Appendix/Glossary`

C

CLI

Short for Command Line Interface. A text based environment that gives you control over your system.

Context

- The circumstances in which an event occurs; a setting.
- The part of a text or statement that surrounds a particular word or passage and determines its meaning.

D

DDD

Short for Domain Driven Design. (from GlossaryTech) Domain-Driven Design is a set of principles and schemes aimed at creating optimal systems of objects. Reduced to the creation of software abstractions, which are called models of subject areas. These models include business logic that establishes a link between the real conditions of the products application area and the code.

I

IDE

Short for Integrated Development Environment. An application that helps you to develop software, by combining useful features and libraries into one single application. These IDEs usually allow you to run your tests and code without needing to leave the comfort of your development environment.

J

Jargon

"The specialized language of a trade, profession, or similar group, especially when viewed as difficult to understand by outsiders.". Its goal is to make the exchange of information more efficient by giving specific names to things that are relevant to the in-group. It is said that the Inuit have over thirty words to differentiate between different types of snow. Other professions, such as software developers, strongly rely on metaphors to refer to technical concepts.

K

KPI

Short for Key Performance Indicator. The critical (key) indicators of progress toward an intended result. KPIs provide a focus for strategic and operational improvement, create an analytical basis for decision-making and help focus attention on what matters most.

P

Paradigm

A *"paradigm"* is a fancy word for "point of view". It is the belief system that you hold and use to make sense of what is happening around you. You can look at it as walking around in an unknown city, armed with a town map. If the map is not sufficiently detailed, you will find yourself ending up in the wrong place. You might even end up driving into a lake if the map is inaccurate, digital, and talks to you.

Pattern language

from wikipedia: A pattern language is an organized and coherent set of patterns, each describing a problem, and the core of a solution that can be used in many ways within a specific field of expertise.

Practices

Practices are a set of actionable recipes, that can help you achieve certain goals. As with all advice, these are not guaranteed to give results. There exists no such thing as a sure-fire approach that works in any situation. To make these patterns as helpful as possible, they include a short description of the circumstances in which they usually work well.

Pragmatic

The word *'Pragmatic'* originally means "skilled in business". You can interpret this as thinking about the added benefit (return on investment) of an action before deciding to do it. A pragmatist will take pieces from various toolsets and methodologies, and apply them to the problem at hand only if it makes sense to use them. This means that even if a new software architecture is hip and trendy, you would look at the issue you are trying to solve first and see if the new approach is worth doing.

R

ROI

Short for 'return on investment'. A concept originating from the financial sector. Described on this page.

S

Synthesis

A term that is used to describe the act of combining (often diverse) conceptions and observations into a coherent whole.

Learning_Materials

`/X_Appendix/Learning_Materials`

There are so many awesome learning materials that the simple act of finding something to read is challenging in and of itself. The list below is a curated index of knowledge resources, including but not limited to:

- books
- articles
- videos
- websites and tools

In order to help you find the next thing to research, we included some symbols next to each resource. The meaning of which can be found in the table below.

Symbol	Meaning
:fas fa-chess-pawn:	Excellent introductory resource
:fas fa-user-ninja:	Practical, and usually opinionated. Pay attention to the context.
:fas fa-trophy:	These resources are exceptionally valuable, and considered *'must read/watch'* by many
:fas fa-microscope:	In-depth, somewhat challenging read.
:fas fa-jet-fighter:	Quick paced, easily digestible.

Shortlist

> A curated list of recommended learning resources

Books && Long-form publications

Name	Author	Tags
The Pragmatic Programmer, 20th Anniversary Edition: your journey to mastery	Dave Thomas, Andy Hunt	:fas fa-trophy: :fas fa-user-ninja: :fas fa-jet-fighter:
Pragmatic Thinking and Learning: Refactor Your Wetware	Andy Hunt	:fas fa-chess-pawn: :fas fa-trophy: :fas fa-jet-fighter:
New Programmer's Survival Manual: Navigate Your	Josh Carter	:fas fa-chess-pawn: :fas fa-user-ninja:

Workplace, Cube Farm, or Startup		
The 7 Habits of Highly Effective People	Stephen R. Covey	:fas fa-trophy:
Behind Closed Doors: Secrets of Great Management	Johanna Rothman, Esther Derby	:fas fa-user-ninja: :fas fa-jet-fighter:
Java by Comparison: Become a Java Craftsman in 70 Examples	Dr. Simon Harrer, Dr. Jörg Lenhard, Linus Dietz	:fas fa-chess-pawn: :fas fa-user-ninja:
Getting Things Done: The Art of Stress-Free Productivity	David Allen	:fas fa-chess-pawn: :fas fa-trophy: :fas fa-user-ninja:
The Passionate Programmer (2nd edition): Creating a Remarkable Career in Software Development	Chad Fowler	:fa fa-chess-pawn:
The Developer's Code: What Real Programmers Do	Ka Wai Cheung	:fas fa-chess-pawn: :fas fa-user-ninja:
The Healthy Programmer: Get Fit, Feel Better, and Keep Coding	Joe Kutner	:fas fa-user-ninja: :fas fa-jet-fighter:
Practices of an Agile Developer	Andy Hunt, Dr. Venkat Subramaniam	:fas fa-user-ninja: :fas fa-jet-fighter:
Nonviolent Communication: A Language of Life	Marshall B. Rosenberg	:fas fa-trophy: :fas fa-chess-pawn: :fas fa-jet-fighter:
Apprenticeship Patterns	Dave Hoover	:fas fa-trophy: :fas fa-pawn: :fas fa-user-ninja:
Programming Kotlin: Create Elegant, Expressive, and Performant JVM and Android Applications	Dr. Venkat Subramaniam	:fas fa-chess-pawn: :fas fa-user-ninja:
Design and Build Great Web APIs: Robust, Reliable, and Resilient	Mike Amundsen	:fas fa-microscope:
Pragmatic Guide to Sass 3: Tame the Modern Style Sheet	Hampton Lintorn Catlin, Michael Lintorn Catlin	:fas fa-chess-pawn: :fas fa-user-ninja: }
Practical Microservices: Build Event-Driven Architectures with Event Sourcing and CQRS	Ethan Garofolo	:fas fa-microscope: :fas fa-user-ninja:
Data Science Essentials in Python	Dmitry Zinoviev	

Exercises for Programmers: 57 Challenges to Develop Your Coding Skills	Brian P. Hogan	:fas fa-chess-pawn: :fas fa-user-ninja:
Semantic Software Design	Eben Hewitt	:fas fa-microscope:
97 Things Every Java Programmer Should Know	edited by: Kevlin Henney, Trisha Gee	:fas fa-user-ninja: :fas fa-jet-fighter:
The Nature of Software Development: Keep It Simple, Make It Valuable, Build It Piece by Piece	Ron Jeffries	:fas fa-chess-pawn: :fas fa-trophy: :fas fa-user-ninja:
Software Design X-Rays: Fix Technical Debt with Behavioral Code Analysis	Adam Tornhill	:fas fa-microscope:
Small, Sharp Software Tools: Harness the Combinatoric Power of Command-Line Tools and Utilities	Brian Hogan	:fas fa-user-ninja:
First Things First	Stephen R. Covey, A. Roger Merrill, Rebecca R. Merrill	:fas fa-user-ninja:
The 8th Habit: From Effectiveness to Greatness	Stephen R. Covey	:fas fa-trophy: :fas fa-user-ninja: :fas fa-jet-fighter:
Principle-Centered Leadership	Stephen R. Covey	:fas fa-user-ninja:
The Five Dysfunctions of a Team: A Leadership Fable	Patrick Lencioni	:fas fa-trophy: :fas fa-user-ninja:
Domain-Driven Design: Tackling Complexity in the Heart of Software	Eric Evans	:fas fa-chess-pawn: :fas fa-microscope:

Articles && Short-form publications

Name	Author	Tags
The Big Refactoring	Jan Van Ryswyck	:fas fa-user-chess-pawn:
16 misconceptions about Waterfall	Michael Küsters	:fas fa-chess-pawn: :fas fa-user-ninja:
Product Manager vs Product Owner	Melissa Perri	:fas fa-microscope:
The 2020 Scrum Guide and Related Content	Scrum Alliance	:fas fa-trophy:
Improving Feedback Flows in	Philipp Hauer	:fas fa-chess-pawn: :fas fa-user-

Organizations with 'Complete Peer Feedback'		ninja:
How to Make Your Development Department More Productive	Dieter Jordens	:fas fa-pawn: :fas fa-user-ninja:
A Leader’s Framework for Decision Making	David J. Snowden && Mary E. Boone	:fas fa-microscope: :fas fa-trophy:
Dilemmas in a general theory of planning [paper]	Horst W. J. Rittel && Melvin M. Webber	:fas fa-microscope:
Getting Serious About Diversity: Enough Already with the Business Case	David A. Thomas	:fas fa-ninja:

Talks

Name	Author	Tags
Beauty in Code 2020: "Getting Back To The Heart of Agile"	Alistair Cockburn	:fas fa-trophy:
Beauty in Code 2018: "Software professionals, we keep using that word..."	Louis Hansen	:fas fa-user-ninja:
Beauty in Code 2020: "Lambda? You Keep Using That Letter"	Kevlin Henney	:fas fa-trophy: :fas fa-microscope:
DDD Europe 2020: Bounded Contexts for Team Organization	Cyrille Martraire	:fas fa-ninja:
Agile Essentials	R. Van Solingen	:fas fa-ninja: :fas fa-trophy:

Blogs and Frameworks

Name	Author	Tags
The C4 model for visualising software architecture	Simon Brown	:fas fa-trophy: :fas fa-jet-fighter:
Sociocracy 3.0: Evolve Effective Collaboration At Any Scale	Bernhard Bockelbrink, James Priest and Liliana David	:fas fa-trophy: :fas fa-user-ninja: :fas fa-microscope:
GROWS Method®: forming good habits in software development	GROWS Method® Institute, Andy Hunt	:fas fa-user-ninja:

Manifesto for agile software development	Various authors	:fas fa-trophy: :fas fa-user-ninja: :fas fa-jet-fighter:
"Software craftsmanship manifesto"	Various authors	:fas fa-user-ninja:
CYNEFIN framework for complexity	Dave Snowden, et al.	:fas fa-microscope:
Coding is like cooking	Emily Bache	:fas fa-user-ninja:
DSL Catalog	Martin Fowler	:fas fa-chess-pawn: :fas fa-trophy: :fas fa-user-ninja:
A Pattern Language for Pattern Writing	Gerard Meszaros, Object Systems Group	:fas fa-microscope:
Memetics	Michele Coscia	:fas fa-chess-pawn:

Repositories and Tools

Name	Author	Tags
GlossaryTech's Glossary: Technology terms	GlossaryTech Team	:fas fa-chess-pawn: :fas fa-trophy:
DDD utilities and info	GlossaryTech Team	:fas fa-chess-pawn: :fas fa-trophy:
NeoVim dotfiles	NeoVim dotfiles - github topic	:fas fa-microscope:
Firefly 3 - finance management	Self-hosted open-source personal finance manager	:fas fa-trophy: :fas fa-microscope:
Structurizr modeling DSL	Structurizr Ltd	:fas fa-trophy: :fas fa-microscope: :fas fa-user-ninja:

Podcasts

Name	Author	Tags
A bit of optimism	Simon Sinek	:fas fa-user-ninja:
Beyond Coding	Patrick Akil	
The education game	Matt Barnes && Dr. Scott Van Beck	
What you will learn	Adam Ashton && Adam Jones	
Scrum master toolbox	Vasco Duarte	
Troubleshooting Agile	Jeffrey Fredrick && Douglas Squirrel	
The productivity show	Asian Efficiency	