Penguin Programming Patterns

A PRAGMATIC GUIDE TO SOFTWARE DEVELOPMENT

DEJONGH STIJN
software development

Author(s):	Stijn Dejongh

Date:	2021-02-14
Document version:	1.0.0-DRAFT
Customer:	public domain knowledge

Contents

1 Document Overview

1.1 Purpose of the document

1.2 Intended Audience

1.3 Document Structure

2 Productivity

2.1 Productivity Patterns

2.1.1 Patterns Overview

This section is a collection of Productivity related patterns. The patterns are meant as a guideline for your day-to-day development activities and will hopefully offer you a mental framework to reason about the tasks you are asked to perform.

For consistency, the patterns follow a similar structure. As we all know, **context matters**. This is why each pattern is prefaced with a short description of when it can be useful to consider using it. The aim is not to apply as many of them as you can on any given task. **This is not a bingo chart.**

The pattern section is followed by a series of **articles** that were written in the past by myself, or others. If I do not hold the copyright of the material included here, it will be mentioned and the original author will be asked for approval before the content is included here.

2.1.2 List of Patterns

Pattern	Pattern type	Description
ROI metrics	Method	Keep track of cost/benefits ratios and apply these to your work and reporting

2.1.3 [Pattern] Return on investment

2.1.3.1 Applicable 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 of the 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 someones 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. It's 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.

2.1.3.2 Description of Pattern 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.

2.1.3.3 Key Performance Metrics

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

2.1.3.4 Related Patterns and Resources

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.

2.2 Productivity Articles

This section contains 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.

2.2.1 List of Articles

Article	Type	Description
Agility and Craftsmanship	Opinion piece	This article is more of a reflection on observations that I have made over the years. It includes a brief history of the agile software development movement, and how these ideas and concepts are often deluted when they are applied naively in real development teams with actual people.

2.2.1.1 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.

2.2.1.2 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.

2.2.1.2.1 The origin 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.

2.2.1.2.2 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.

2.2.1.3 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.

2.2.1.3.1 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.

2.2.1.4 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.

2.3 A Pragmatic Paradigm

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.

2.3.0.0.1 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

Skill Level/Mental Function	Novice	Advanced Beginner	Competence	Proficient	Expert
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.

2.3.0.0.2 Knowledge to take with you 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 look-out for improvements to your process. But most of all, care deeply about what you are doing.

2.3.0.1 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.

2.3.0.2 Appendix

2.3.0.2.1 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

2.3.0.2.2 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

author	url	title	publisher
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 the Aspiring Software Craftsman	O’Reilly Media, Inc
van Solingen, R	talk on youtube	De Kracht van Agile (8 ankerpunten voor de praktijk)	Agile 2019 talks - distributed on youtube.com

3 Leadership

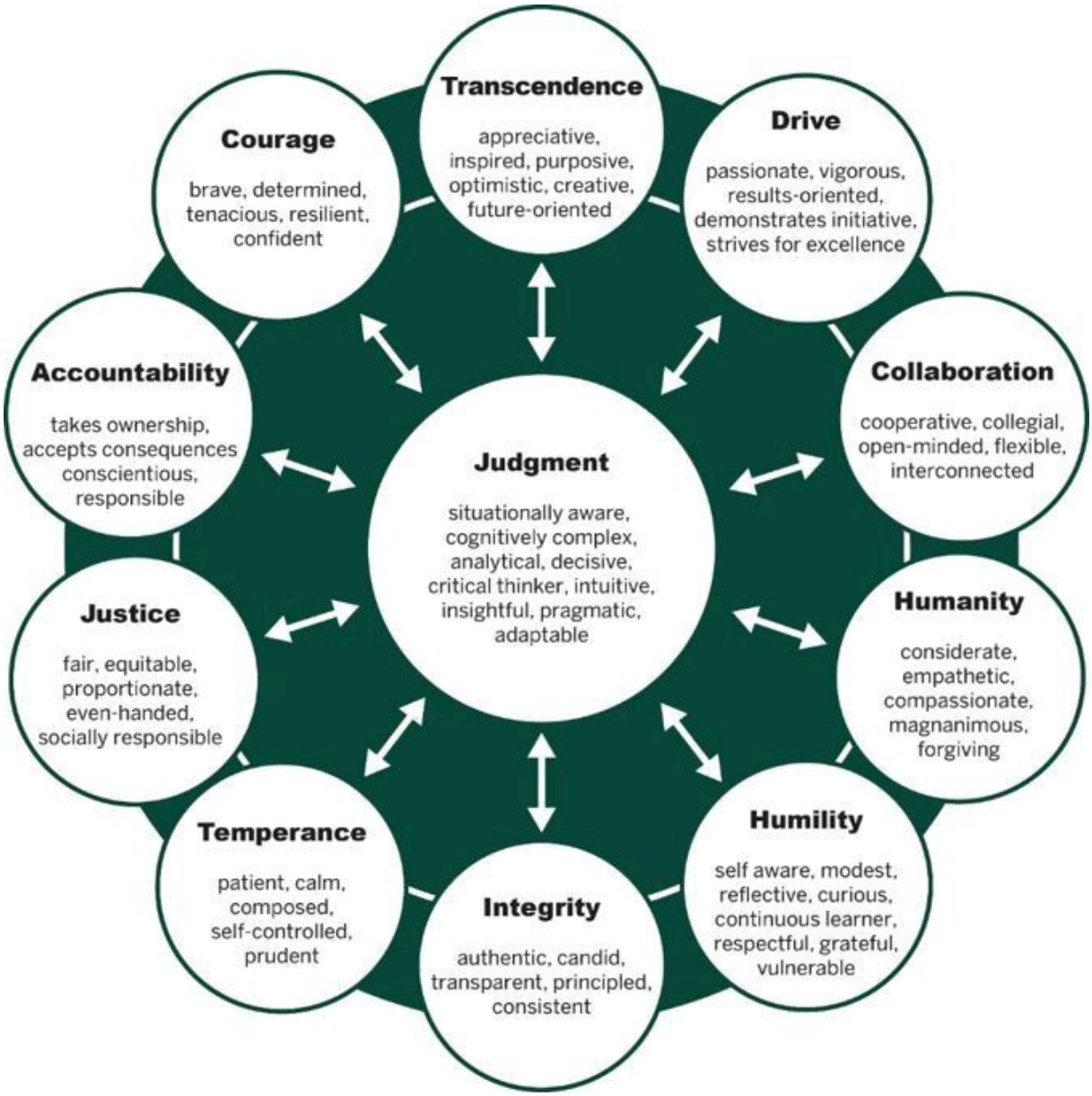

Figure 1: Depiction of the core values of leadership

3.1 Further reading

4 Programming Patterns

4.1 Overview

This section is a collection of Programming related patterns. The patterns are meant as a guideline for your day-to-day development activities and will hopefully offer you a mental framework to reason about the tasks you are asked to perform.

For consistency, the patterns follow a similar structure. As we all know, **context matters**. This is why each pattern is prefaced with a short description of when it can be useful to consider using it. The aim is not to apply as many of them as you can on any given task. **This is not a bingo chart.**

4.2 Why does code quality matters?

4.2.1 Introduction to clean coding

> You should 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 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.

4.2.2 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 because the code is plain **hard to understand**. It is usually not expressive enough, or very verbose.

> Code is write once, but read very often.

A good measure to know whether or not the code you see before you is *clean* enough is to use the infamous *"WTFs per minute metric"*. This metric originates from a well-known cartoon that has been recreated multiple times.

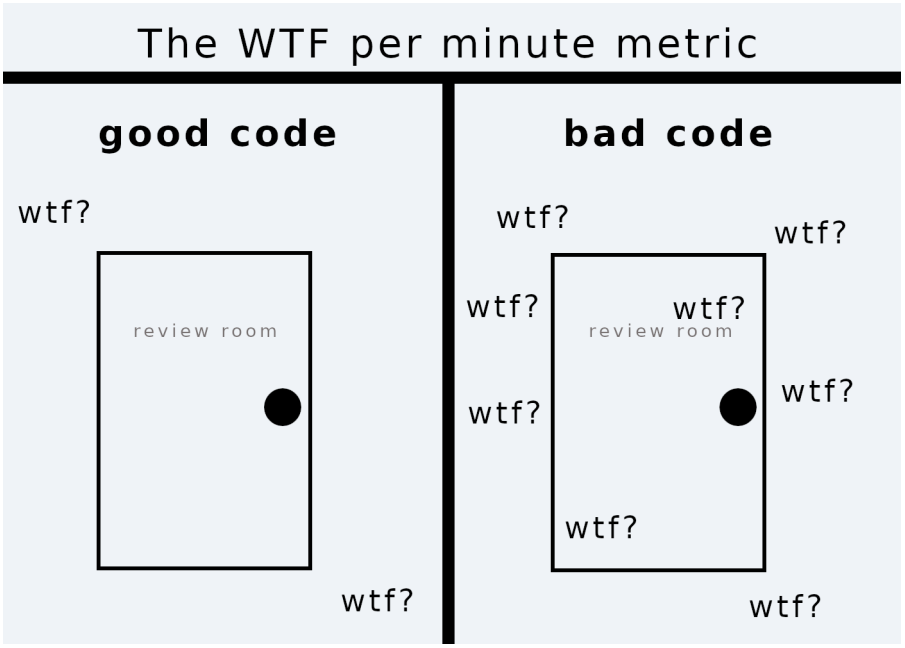

Figure 2: my attempt at reproducing the famous cartoon

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.

> People prefer pretty things

4.3 Section Contents

4.3.1 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 writting a variety of languages, but the default language is Oracle Java. The table below can assist you in downloading the development environments you need to work with the included code samples.

Technology	Download link
Java Development Kit	Oracle website

4.3.2 List of Patterns

Pattern	Description
Good Enough Code	Avoiding the urge to overdesign your code by sticking to a good-enough implementation

4.4 Baptizing your code

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.

4.4.1 Applicable Context

You are writing or reading code written by yourself or colleagues, and notice that some of the function names or parameters have names that do not add much meaning to the code itself. 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.

Some of the thoughts that are likely to cross your mind while reading the code are:
- [ ] *"I think I kind of know what this code block is supposed to do"*
- [ ] *"So this variable contains the result of the calculation, right?"*
- [ ] *"We start in method X which does a thing, and then call this other thingy that does ...."*

4.4.2 Description of Pattern

4.4.3 Key Performance Metrics

4.4.4 Related Patterns and Resources

Item	Description	Action
Some thingy	Why it is here	What to do with it?

4.5 Good Enough Code

4.5.1 Applicable Context

Issue: You get told that you overcomplicate simple tasks.

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).

Writing clean code is admirable, but it also has to make sense for the problem at hand. Creating a specific design by applying a pattern is to be done when it solves the problem at hand and makes the code more readable, robust, extensible or reusable.

Figure 3: Sometimes it is okay to keep it simple

4.5.2 Description of Pattern

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 fixed easily at that time?"*

The idea of Good Enough Code is to write code as well designed as it needs to be AT THIS POINT IN TIME. If an idea for a more generic solution comes to mind during your implementation, take note of it. If in the future the problem you anticipated actually happens, or the code you wrote now is reused, it will be solved at that time.

Make sure the code you write at this point in time adheres to the basic principles of clean code and design, but do not solve future problems that might never happen.

4.5.3 Key Performance Metrics

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

4.5.4 Related Patterns and Resources

Item	Description	Action
Enterprise Quality FizzBuzz	A prime example of overdesigning 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. Appart from learning how he thinks, the excercise also focusses on supressing your personal need to overly beautify a simple project.	Do the excercise and stop yourself from creating too many classes. Repeat the mantra: *"This is fine for now"* to supress your urges to add indirection or OO concepts to the design.

5 Software Architecture

5.1 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.

5.1.1 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.

5.1.2 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.

5.2 Architecural Patterns

6 Glossary

6.1 Terminology, Acronyms and Definitions

Term	Acronym	Definition
Integrated Development Environement	IDE	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 environement.