THE BENEFITS OF UNDERSTANDING THE WHYS BEHIND THE HOWS

REBECCA WIRFS-BROCK, Wirfs-Brock Associates

This essay reflects on how pattern descriptions may not be so easily understood and what we might do to improve this situation. When reading new-to-them software patterns, experts view those patterns through the unique lens of their experience. Patterns aren't mechanically implemented; they are carefully woven into a design. Expert designers produce patterned solutions that fit with their perceptions of their design environment. Since most software developers don't share experts' underlying design values, practices, and principles, there's quite a disconnect between what is said in pattern descriptions and which is perceived and understood by pattern readers. If we learn patterns along with some design principles and practices (and some rudimentary understanding of why these principles and practices are important), we can become equipped with knowledge that allows us to make connections between these principles and how they are applied.

Categories and Subject Descriptors: • Software and its engineering~Software design engineering • Software and its engineering~Design patterns

ACM Reference Format:

Wirfs-Brock, R. 29th Conference on Pattern Languages of Programming (PLoP), PLoP 2022, Oct 12 -16, 2022, xx pages.

1. INTRODUCTION

Experienced software designers quickly grok the essence of software design or architecture patterns. Subsequently, they are able to craft reasonable design solutions based on the (scant) information they find there.

How do they do this?

Experts have accumulated a wealth of tacit knowledge about how to design and effectively build software. Experts nudge their design in directions they want it to go, utilizing patterns along with countless (unnamed) heuristics according to their personal design aesthetics. When reading new-to-them software patterns, they view those patterns through the unique lens of their experience. They implicitly add and enrich pattern descriptions based on their umwelt.

Umwelt, from the German, means "environment" or "surroundings:

> "…for everything that a subject perceives belongs to his perceptual world [merkwelt] and everything it produces, to its effect world [wirkwelt]. These two worlds, of perception and production of effects, form one closed unit, the environment [or umwelt]" [vU].

It is the experienced designer's umwelt that empowers them to apply patterns holistically. Deeply ingrained personal heuristics, values, and design principles and practices play a central role in employing patterns in a design. Patterns aren't mechanically implemented; they are carefully woven into a design. Expert designers produce patterned solutions that fit with their perceptions of their designed environment. All the while making larger or smaller design adjustments based on subtle cues and gut feelings.

In sharp contrast, non-expert designers may not pick up on salient cues in their design environment. They don't readily spot those cues that could help them shape or adapt a pattern to their current design. They may not even be aware of what to look for. While they might grasp the essence of a particular design pattern, they aren't likely to fully understand its significance or impact. They are unlikely to anticipate the constraints certain patterns place on future design choices. Nor do they relate that pattern to other existing patterns or structures in their design (whether named or known by them as patterns or not). They have a very different umwelt than expert designers and pattern authors.

As a consequence, simply learning more design patterns doesn't guarantee that a less-skilled designer becomes a better designer, let alone an expert.

The body of software design and architecture patterns have mostly been written by experts. While these experts may believe they are describing their patterns in sufficient detail so that designers who do not share their umwelt can pick up and use them, I'm doubtful.[1] Since most software developers don't share experts' underlying design values, practices, and principles, there's quite a disconnect between what is said in pattern descriptions and which is perceived and understood by pattern readers.[2]

2. ACQUIRING JUDGMENT AND DISCERNMENT

For several years I've explored Billy Vaughn Koen's definition of heuristics and their relation to software design practices, and software design and architecture patterns [Wirf17, Wirf18, WirfWK, Wirf20]. I've written blog posts and essays and given talks, keynotes, and workshops about heuristics (for a gentle introduction to different kinds of heuristics see Growing Your Personal Design Heuristics Toolkit [Wirf19a]).

My goal has been to make software designers more aware of their own personal design heuristics and to acknowledge their value. Furthermore, I caution them to not take every bit of design advice they find as authoritative, but to question whether specific advice—including software patterns—is appropriate to their design context.

In *Discussion of the Method: Conducting the Engineer's approach to problem solving* [Koen], Billy Vaughn Koen defines a heuristic as, "anything that provides a plausible aid or direction in the solution of a problem but is in the final analysis unjustified, incapable of justification, and potentially fallible." Vaughn Koen identifies three kinds: heuristics that lead you to take a specific action (design patterns are such kinds of heuristics, but there are many other heuristics larger and smaller than individual design patterns), context-sensitive heuristics that shape your beliefs and values, and heuristics that guide you in your use of other heuristics (for example, pattern languages suggest paths and sequences of connected actions).

I begin presentations by telling the story of cooking my very first Blue Apron[3] recipe for Za'atar Roasted Broccoli Salad (for details of this story see the blog post, Nothing Ever Goes Exactly by the Book [Wirf19b]. I jokingly pointed out all the places where the recipe suggests to add salt. I postulated that if I followed Blue Apron instructions without applying judgment, the dish would be too salty to eat.

I shared how I applied my own judgments to modify the instructions to fit with my understanding of what makes for a tasty, not-too-salty dish. Instead of mindlessly following the recipe. In summary: I ignored several places where the recipe suggested adding salt as well as made personal interpretations of other imprecise instructions (such as determining the appropriate amount of liquid for a "drizzle").

One heuristic I used was: "ignore instructions for adding salt if it **seems** (based on my gut feel) to be excessive." Also, "only add salt to taste at a step where you can actually taste its effect." Following those heuristics, I made a blander dish that, while looking great, lacked flavor. Why? Because, as I found out later, adding salt to a dish at the end only salts the top layer of food (and unevenly at that).

But… achieving a tasty dish wasn't the point of my story! I wanted to get people fired up and more confident with their own judgments. They should feel comfortable applying knowledge gained through previous software design experiences to new situations. Each of us have valuable insights and experiences to draw upon.

With my story, I was trying to illustrate the application of a general heuristic—think for yourself—without understanding the umwelt of the situation (cooking). In hindsight, I misled my audience by painting an overly simplistic picture of software design, design expertise, and exercising judgment. Simply trusting and mindlessly following "recipes" or "patterns" because they are published or credentialed might well lead to inedible dishes or to poorly designed software. While we should value and treasure our experiences, reflect on them, and draw upon our heuristics more often, we should also be cautious about their relevance in new

[1] Pattern descriptions are intended to illustrate or describe the essential aspects of a pattern, not to instruct designers on how to apply them.

[2] Alexander's patterns in *A Pattern Language* (APL) [AISJFA], were also written by experienced, thoughtful building architects who hoped their patterns would be useful to less-experienced architects (or even laypersons). Although APL makes for good reading, as a layperson, I find I am only able to spot them when I encounter them, or to bring them up in conversations with architects versed in APL. Being able to design a well-architected building or town using these patterns is quite another matter.

[3] Blue Apron is a service that sends you all of the ingredients and instructions to cook a meal at home. You choose from their meals for the week, and you get the ingredients needed for that meal delivered in a box on your doorstep.

situations. We are misleading ourselves if we think we know best what to do in situations where we're not grounded with sufficient knowledge and relevant experience.

3. KNOWING WHY THINGS WORK ENABLES US TO MAKE ADAPTATIONS

Last December I read *How to Taste* [Sel], by Becky Selengut. It's a book on how to become a better cook, by becoming more informed about both facts about taste and potential compensating actions to alter taste. The introductory chapter starts:

> *"Telling you to 'season to taste' does nothing to teach you how to taste—and that is precisely the lofty goal of this book. Once you know the most common culprits when your dish is out of whack, you'll save tons of time spinning your wheels grabbing for random solutions. You'll start thinking like a chef. Some people are born knowing how to do this—they are few and far between and most likely have more Michelin stars that you or I; the rest of us need to be taught. I've got your back."*

That grabbed my attention!

Unless I was superhuman, I can't rely on instincts alone to cook. If I follow recipes without knowing why they work, I simply become good at following recipes. My umwelt has grown with my ability to follow recipes. But I haven't learned ways to "tune" or "adapt" a recipe according to my personal tastes or to use the ingredients I have at hand. Furthermore, I won't know what to do to get back on track when things don't go as expected.

My experiences with cooking have largely been unstructured—dare I say random. Until recently, I had no theory behind what makes a good dish good. I have never taken a class on how to cook or bake. I'm a cooking autodidact.

My heuristics for salting that Blue Apron dish came from who knows where. Perhaps I observed my mother not using much salt in her hastily cooked meals. We always had salt at the table though. Once I started cooking on my own, I followed recipes unless they were too difficult or, in my naïve estimation, had unnecessary steps. I rarely learned why I was instructed to do some things and why the steps were in a particular order. I didn't think deeply about the recipe or how to improve on it, let alone why it worked. Pretty much, I followed instructions while trying to be efficient. If there were two steps that could obviously be done at the same time, I'd do them. No need to boil the water before measuring out the noodles. Over time, I have learned more shortcuts and substitutions. I can modify recipes in limited ways to use the ingredients I have at hand, or perhaps to lower their fat content. My knowledge on how to do so and why it works has largely been acquired through searching the internet. And while my cooking techniques have become more efficient, I haven't developed the ability to craft a dish with nuanced flavors, let alone improvise much. (I'm getting better at this, thanks to Becky's advice and several other books that are teaching me about complementary flavors).

Becky suggests to read her book "…start[ing] at the beginning, as I intend to build upon the concepts one puzzle piece at a time." Each chapter presents fundamental facts, reinforced by a recipe that highlights the important points of the chapter and then suggests Experiment Time activities intended to develop our palate. After we learn basic principles of taste, we learn about salt, acid, sweet, fat, bitter, umami, aromatics, bite, and texture in that order.

Aha!

One way to learn how to exercise judgment is to perform structured experiments after being presented with bit of theory about why things work the way they do—in this case, the chemistry of cooking.

I quickly read through the chapter on salt. Salt is a flavorant—bringing out the flavor of other ingredients. Salting *early* and *often* can dramatically improve the taste of a dish. Salting onions as they sauté not only speeds up the cooking process by causing them to sweat out water, they also become sweeter as a result. When you only salt soup at the end, no matter how much you add, the flavors of unsalted ingredients (for example potatoes), will fall flat. You can end up over salting the stock and still have tasteless, bland potatoes in your soup. Salt needs to be added at the right time, often at several steps in the cooking process, to enhance the flavors of individual ingredients. Ingredients need to absorb salt to enhance their flavor at the right time. Also, to my delight, I learned that different kinds of salt—iodized, kosher, flaky, fine-grained sea salt, each have their own flavoring properties and ratios in recipes. Inspired by Becky's writing, I painstakingly made her chicken soup recipe in the back of the chapter, taking over two days to finish the soup. It was stunningly delicious!

This also made me rethink my previous Blue Apron cooking experience.

4. THE PROBLEM IN A NUTSHELL?

Blue Apron's pretty pictures, step-by-step instructions, and online videos do little to help me to understand how to achieve tasty dishes of my own creation or why certain flavor combinations work. Blue Apron doesn't have bad recipes, they simply aren't designed to teach me anything. In the process of preparing a dish I might learn of some ingredients that I could incorporate in other dishes. But I won't know *how* to do that. My learning is incidental.

That's not a problem if all I want to do is cook a passable meal following a recipe using supplied ingredients. It becomes a problem when I want to get better at cooking tasty dishes on my own. While I do learn some generally useful skills—like how to roast vegetables or mash potatoes, I don't gain deeper knowledge that allows me to improvise. Their recipes focus on how to make a dish efficiently—not why it tastes the way it does or how I might alter it to suit my preferences (to be fair, that's not entirely accurate, they do caution you to carefully add spicy ingredients in small doses to taste).

I find that much information we software designers absorb—whether about design practices, patterns, or techniques is either presented as step-by-step lists of instructions or guidelines—without accompanying explanations of why it makes sense to do so—or as discrete facts. When we join an existing development team working on some new-to-us system, we tend to go with the flow (and the existing software artifacts without any deep understanding of its umwelt or the values and beliefs of its initial designers). Over time we may learn these things, but our learning, is mostly ad hoc and unstructured.

With patterns, this is also the case. Some pattern descriptions present us with a few considerations that could help us determine whether a pattern might a "reasonable fit."[4] That is, if we share enough of a design umwelt with the pattern authors. But patterns never come with explicit instructions on how to adapt a pattern to our specific design context. In design, as with cooking, some steps (or aspects of pattern solutions) are more rigorous or prescriptive than others. For example, if I don't let bread dough rise long enough, I'll end up with a hockey puck instead of a well-formed boule. Similarly, if I don't define a common API for all related Strategy [Gamma] objects, I won't get the benefit of them being interchangeable. Rarely are we warned of consequences of not following a particular step in a pattern solution to the letter. Sometimes details might be important; other times, not so much.

In a recent email exchange with Chris Richardson (author of a book on Microservice Patterns [Rich] and organizer of an online community on that architecture topic) Chris posed the question:

Hi,

I hope everyone is well.

I had a random thought about the Microservice Architecture pattern.

Perhaps, like other architectural patterns, it's very high-level: architect the application as a set of loosely-coupled services. That's quite different from "design patterns" where the participants [elements in the design] typically have clearly defined responsibilities.

Moreover, there's no guarantee that simply breaking your system up as a set of services achieves the benefits: i.e. the resulting context is not guaranteed.

Has anyone written about the topic of 'vague patterns'.

Just wondering.

Chris

I replied:

Chris

[4] To be fair, software design pattern descriptions often briefly discuss some "forces" we should balance in our design and the context in which the pattern is best suited. Summary descriptions of potential consequences of applying the pattern are sometimes present. But these brief descriptions assume the reader has the necessary background to relate to these descriptions to their own situation, and understand their significance.

I think that topic (vague patterns) would be a good one to tackle. Patterns in isolation aren't that interesting.

It is important to weave related patterns into a pattern language that guides you how to solve your problem.

What is the problem that a microservice architecture is trying to solve: decoupling parts of a complex system, allowing them to scale in performance independently, etc.

But what makes a microservice architecture a "good" design choice? Paying attention to localize dependencies, provide coherent behaviors (I think), and to have the ability to evolve microservices semi-autonomously….but still, this begs the question: what makes a microservice architecture implementation a good one.

On the other hand, I think the patterns at the GOF level are like bricks. Nothing guarantees you can assemble a bunch of bricks and make a good building. There's more to it than that. A series of design choices around control, coordination, the ability to evolve and extend the design, as well as necessary flexibility.

Just a few thoughts….

Rebecca

My initial reaction to Chris's question was that patterns—at least the software and software architecture patterns I'm aware of—are written at many different levels in a design. Some are about larger structuring or overall organization of a complex software system into cohesive parts (for example, the Microservice pattern is an architecture style; as is the Hexagonal Architecture pattern [Co]). These patterns are "vague" in that they don't prescribe how to create such structures given the current state of your system. Consequently, there is no guarantee that applying these patterns will result in any improvement. Quite simply, they don't offer recipe-like solutions.[5] Other software patterns are about much smaller design considerations (really coding considerations), for example how to hide implementation behind a Façade [Gamma] or how to define plug-replaceable behaviors using the Strategy pattern [Gamma]. When you use such lower-level design "bricks" you aren't attempting to design a sound structure (it takes much more than a single brick to build a wall and a single pattern to build a software system), but instead to create a well-defined small design element that will fit into a coherent design (along with myriad other "bricks", whether they are applications of patterns or not). Usually, descriptions for these low-level patterns include sample solution sketches that appear at first gland to be templates or recipes (e.g., here's a stylized depiction of what a good structure and behavior should look like…now you go about implementing it with your design elements). Software architecture patterns that define an overall structure for a system, on the other hand, don't come with foolproof recipe-like or templated solutions.

As Joe Yoder points out at the end of our email thread, there are no guarantees of design success when applying either large-scale structural patterns or lower-level design patterns. Architecting takes care, design skills, experience, and as Joe says, "a lot of them come down to good domain modelling principles":

Hi Chris:

Good to hear from you and I hope you are doing well. Rebecca makes a lot of good points that I also subscribe to.

And you make a very good point "there's no guarantee that simply breaking your system up as a set of services achieves the benefits". That is why I like one of your early patterns where you consider monolith vs microservices. A monolith is not an anti-pattern. Also, even if the environment is right for microservice, if you don't pay attention and do good modelling, you will not get the benefits and can have a worse problem.

[5] Recently, Joseph Yoder and Paulo Merson wrote patterns about approaches to evolving an existing system architecture to use microservices [YM]. These patterns are promising in that they illustrate alternate system design paths for re-architecting a system or adding new functionality. Reading these patterns won't make you expert at microservice architecture design, but they do a credible job at distilling these experts' wisdom and experiences in pattern form.

I don't know of anyone who has written vague patterns. I have tried to outline some design principles for microservices. A lot of them come down to good domain modelling principles.

-Joe

Consequently, if we learn individual design techniques or individual patterns, without learning some good design principles and practices (and some rudimentary understanding of why these principles and practices are important), we won't be able fill in the gaps or make connections between these principles and how they apply to these patterns.

Without such grounding, we are tempted to interpret lower-level software design patterns as instructions to be followed because someone (the pattern author) authoritatively says this is what to do. Over time we may build up a playbook of various procedures, design patterns, and design practices but our understanding of why they work (or what alternatives there might be) won't be deep, or rich, or adaptable.

Unless facts are connected to existing knowledge, then we won't necessarily be able to make meaningful adaptations in novel situations. For example, I may learn what to substitute for a missing ingredient in a specific recipe, e.g., substituting oil for butter, but I won't learn about the effects it has in other contexts. I won't know how various fats interact with other ingredients (or going even deeper, why they have these effects, and how different fats work at different temperatures).

Likewise, in software design, I may learn about the latest features of the various React libraries if I am a JavaScript programmer (for example, see https://brainhub.eu/library/top-react-libraries), or common problems and how to overcome them (https://brainhub.eu/library/react-js-problems). But I don't really learn much about the mechanics of rendering or re-rendering web pages, or strategies for handling long lists of items).

To truly gain proficiency in cooking or software design or programming, experiences alongside instruction (or information) that emphasizes the why along with the how is what I need.

Teach me salient facts that ground what I'm about to do. Spark my curiosity. Inspire me. Give me opportunities to tinker and apply newfound knowledge or techniques to see what happens. Only then will new actions and insights become integrated with my umwelt. Even then, I'll need practice to become proficient at using new skills and knowledge.

In hindsight, I believe that the Blue Apron salt story I shared in my presentations wasn't misleading—it just fell short in equipping people with tools and techniques for actively learning and integrating new heuristics into their umwelt.

5. LEARNING TO ADAPT

How does the way we learn relate to becoming more skilled as designers?

Anthropologist Gregory Bateson, developed a theoretical framework for learning "levels" or "types" [Vis, Lutt]. The most basic level of learning is reacting to something, but without learning anything "new". For example, you turn off the oven after the timer beeps. The next level, which he called proto-learning, is learning to change your response in order to correct an error. I learn to take the bread out of the oven, regardless of bake time, if I see the top getting too brown). In this case I'm not blindly following recipe instructions, but through learned observation and perceptions, altering my behavior. The next level of learning, called deutero learning goes a little meta: we learn to adapt our responses based on recognizing and reacting to changing context. For example, I may learn how to adjust timing and baking temperature when baking at altitude. With experience, I'll recognize that I may need to adjust timing and temperature when using an unfamiliar oven. Deutero learning impacts the way we see things, what we anticipate, how we unconsciously behave and what we perceive as "normal" responses in specific contexts. Bateson and others [Vis, Lutt] point out that you shouldn't think of one learning "level" or "type" as being "better" than another, nor that you only are engaged in one type of learning at a time. We're more adept than that.

Learning at various levels happens regardless of our intentions (or awareness). Design habits and heuristics seep in to our umwelt through our experiences. And our experiences beget habits, or as I prefer to think of them, "cherished" or "favored" heuristics.

6. SOME THOUGHTS ON GROWING DESIGN EXPERTISE

While I want people to be confident in drawing upon on their experiences, I also want them to appreciate that some ingrained design habits may be based upon shaky premises that don't hold up in different design context. Falling back on old, familiar heuristics without considering whether they are appropriate in a novel context can result in unpleasant surprises. Worse yet, when we don't know some of whys behind what we are doing, we may not know how to take corrective actions.

There are many situations when it is perfectly fine to act on instinct (or rather, unconscious learned behaviors and knowledge). Being deliberate and intentional all the time isn't possible or desirable.

And yet, actively acquiring and building deeper expertise and expanding your umwelt requires some directed attention.

In my talks I have spoken of the practice of keeping a design journal or daily log of your thoughts and design actions. This harkens back to early in my engineering career when I recorded design notes in my engineering notebook.[6] This record left me some breadcrumbs I could reflect back on. I'm somewhat embarrassed to admit that I didn't do that often.

I also challenged audiences to extract and examine heuristics (and especially competing heuristics) from talks, presentations, and others' writings. But this begs the question, how can they effectively incorporate these new alternative approaches into their own design umwelts?

I have some preliminary thoughts about this, but no easy answers.

I now believe it is equally important to seek the why behind the what you are doing (especially when tackling a novel design problem). Ask: Why does a heuristic or pattern have a desired effect, and why at other times does it fail? What other alternative approaches should I consider? What do others think of my design ideas?

Software design, however, is often done in bits and spurts. Under pressure to crank out production-ready code there is little time for introspection. You are focused on making the design work only enough so that it can be put to use without breaking things. Your focus is narrow.

Only when you pause, can you reflect on why your design works the way it does and what other approaches you might have considered.

Unless you work in an environment where regular slack time, design discussions and experiments are integral to your work rhythms, learning (and widening and deepening of your umwelt) *will* be spotty. Exploration of design options, and reflection can play a big part in growing design skills. As does practice. Woody Zuill, writes how their mob programming team[7]—a small team of individuals all working together on the same code at the same time—approach the team learning together [Zui]:

> *"…we also take time to "re-sharpen the saw" daily by spending the first hour of the day in a group study session. Additionally, we have an extended study session most Fridays to do a more intense study for 2 or 3 hours. In our daily study sessions, we select some aspect of programming that we feel is a weak spot for us, and spend an entire hour studying it. We usually do our study as a workshop and run it as a Coding Dojo similar to our Mob Programming style. We'll use any technique that helps, such as working through a code kata, watching on-line video training, studying a book, or tackling some interesting algorithm or some new technology.*
> *Since we work in very short iterations of a day or two it is easy for us to experiment with various ways to do things. We keep an eye out for any aspect of our work that we can automate or simplify and try any approach that we think might work. This includes both programming and process related ideas. For example, if we have several ideas for solving a problem, but with no clear winner across the team, we'll try a minimal version of each solution and see which we like better. The cost for doing experiments is relatively low, and the payoff for us is often many times the time invested."*

---

[6] Tektronix gave engineers such notebooks with nominal instructions: write about daily about design ideas you thought of, and to be sure to sign and date each page. This was intended to be evidence for patent applications. But these notes inadvertently served another purpose—to remember how your thoughts and design ideas progressed.

[7] Another name for mob programming is ensemble programming. This practice has its roots in agile development, where fast iterations with a steady stream of incoming feature requests are the norm and developers mostly write and design code together rather than working solo.

There are ways to incorporate learning into daily rhythms: Deliberate practice trying out new patterns and heuristics; noodling around with new design approaches to see how they might work; experimenting, just for the sake of gaining experience; reflecting on what you've done; writing down some heuristics you've applied; receiving timely critical feedback and constructive critique, these all help you improve your design skills and broaden your umwelt.

But, rarely do we take the time for such design playfulness. We should make more space for these kinds of activities. But even playful exploration and experimentation might not go far enough.

7. LETTING GO OF CERTAINTY

Sometimes it is important to loosen your grip on those naïve, unarticulated heuristics you may be holding onto and shake up your comfy design worldview (or at the very least temporarily suspend the belief that what you know is all you need to know). Admit that there are design techniques and approaches waiting for your discovery. Some few may have been written up as software design or architecture patterns, others you may find by reading code, blog posts, technical literature, or through experimenting.

To actively reshape, expand and tune your design umwelt you need to break from your routines and dive into actively learning how new design approaches might work to your advantage.

One uncomfortable truth about this umwelt reshaping/expanding activity is that can cause you to shed some cherished heuristics. As you adjust your world view, you may have to let go of some strongly held design beliefs or perceptions (or recalibrate their utility). Breaking out of design "habits" as Woody Zuill's story illustrates, need not be painful, however, when you approach learning with curiosity and playfulness.

8. ACKNOWLEDGEMENTS

I'd like to thank my shepherd, Philipp Bachmann, for reading early drafts of this essay, making constructive suggestions and observations, and prompting me to try harder to clarify what is in this essay for my intended audiences. I'd also like to thank my virtual writers' workshop colleagues …. And finally, thanks to my supportive and consistently thoughtful critic, Allen Wirfs-Brock.

REFERENCES

[AISJFA] Alexander, C., Ishikawa, S., Silverstein, M., Jacobson, M., Fiksdahl-King, I., Angel, S. (1977). *A Pattern Language: Towns, Buildings, Construction.* Oxford University Press.
[Co] Cockburn, A. "Hexagonal Architecture," (2005), blog post, retrieved September 21, 2022: https://alistair.cockburn.us/hexagonal-architecture/
[Gamma] Gamma, E., Helm, R., Johnson, R., Vlissides, J. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1995.
[Koen] Koen, B.V. *Discussion of the method: Conducting the Engineer's approach to problem solving*, Oxford University Press, 2003.
[Lutt] Lutterer, W. "Anthropology of Learning: Gregory Bateson," Encyclopedia of the Sciences of Learning, Heidelberg: Springer, 2012, p. 412-415.
Rich] Richardson, C. *Microservices Patterns: With examples in Java,* Manning, 2018.
[Sel] Selengut, B. *How to Taste: The Curious Cooks Handbook to Seasoning and Balance, from Umami to Acid and Beyond with Recipes,* Sasquatch Books, 2018.
[Vis] Visser, M. "Gregory Bateson on Deutero-learning and Double Bind: A Brief Conceptual History" in Journal of History of the Behavioral Sciences, Vol. 39(3), 269 – 278 Summer 2003 Published online in Wiley InterScience (www.interscience.wiley.com).
DOI: 10.1002/jhbs.10112
[vU] von Uexküll, J. (1934/2010). *A Foray into the Worlds of Animals and Humans with a Theory of Meaning*. Minneapolis, MN: University of Minnesota Press.
[Wirf17] Wirfs-Brock, R. "Are Software Patterns Simply a Handy Way to Package Design Heuristics?" (2017). Proceedings of the 24th Conference on Pattern Languages of Programs (PLoP'17).
[Wirf18] Wirfs-Brock, R. "Traces, tracks, trails, and paths: An Exploration into How We Approach Software Design" (2018). Proceedings of the 25th Conference on Pattern Languages of Programs (PLoP'18).
[Wirf19a] Wirfs-Brock, R. "Growing Your Personal Design Heuristics Toolkit," blog post, retrieved September 21, 2022: https://wirfs-brock.com/blog/2019/03/20/growing-your-personal-design-heuristics/
[Wirf19b] Wirfs-Brock, R. "Nothing Ever Goes Exactly by the Book," blog post, retrieved September 21, 2022: https://wirfs-brock.com/blog/2019/04/19/nothing-ever-goes-exactly-by-the-book/
[WK] Wirfs-Brock, R and Kohls, C. "Elephants, Patterns, and Heuristics." (2019). Proceedings of the 26th Conference on Pattern Languages of Programming (PLoP '19).
[Wirf20] Wirfs-Brock, R. "Should we stop writing design patterns?" (2020). Proceedings of the 27th Conference on Pattern Languages of Programs (PLoP'20).
[YM] Yoder, J. and Merson, P. "Strangler patterns." (2020) Proceedings of the 27th Conference on Pattern Languages of Programs (PLoP'20).
[Zui] Zuill, W. "Mob Programming – A Whole Team Approach." (2014) Agile 2014, retrieved September 21, 2022: https://www.agilealliance.org/resources/experience-reports/mob-programming-agile2014/