OBSERVATIONS ON GROWING A SOFTWARE DESIGN UMWELT

REBECCA WIRFS-BROCK, WIRFS-BROCK ASSOCIATES

Pattern authors ostensibly are experts on the topic their patterns address. Since most software developers don't share those experts' underlying design values, practices, and principles—let alone their design context—there's a disconnect between what is said in pattern descriptions and that which is perceived and understood by pattern readers. As pattern readers, we will be better equipped to grasp the significance of these software patterns if we also gain insights into the design principles and values that they are based on and some reasons why they "work" the way they do. But it is through experience, practice, and reflection, that we deepen the connections between new-to-us patterns and our existing knowledge, and expand our design umwelt.

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 17 - 24, 2022, 10 pages.

1. INTRODUCTION

The body of software design and architecture patterns have mostly been written by experts. While pattern authors may believe they are describing their patterns in sufficient detail such that other software designers can pick up and use them, I remain skeptical. Pattern descriptions by intent only illustrate or describe the essential aspects of a pattern; they do not provide detailed instructions on how to apply them or explain why a pattern "works" to solve a particular design problem. And since most software developers don't share pattern authors' underlying experiences, there's a disconnect between what is said in pattern descriptions and what is perceived and understood by pattern readers.

Yet many 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?

Experienced software designers have accumulated a wealth of tacit knowledge about how to design and effectively build software. They nudge their designs in directions they want them to go, using patterns along with countless (unnamed) personal heuristics. When reading new-to-them software patterns, they view those patterns through the unique lens of their experiences. They implicitly add and enrich sparse 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]" [Ux].

Our umwelt isn't static; we reshape our umwelt as we interact with the world, gaining skills and learning to perceive differently. And it is our umwelt that both drives and limits our potential actions.

It is the experienced designers' umwelt that empowers them to apply patterns holistically. Deeply ingrained personal heuristics, values, design principles, and practices play a central role in how they use patterns in their designs. Experienced designers create 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. They don't mechanically implement patterns; instead, they carefully weave them into their designs, adding what is needed to make the new patterns fit with their designs and satisfy their design aesthetics.

In sharp contrast, inexperienced designers, while they might grasp the essence of a particular design pattern, aren't likely to fully understand its significance or design impact. It is difficult for them to spot salient signals that could help them shape or adapt a pattern to better fit their current design. They won't anticipate the constraints certain patterns place on future design choices. They may struggle to relate that pattern to existing structures in their design (regardless of whether those structures are named or known to them as patterns). They have a very different umwelt than experienced designers and pattern authors.

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

2. HEURISTICS AND DISCERNMENT

Billy Vaughn Koen, in *Discussion of the Method: Conducting the Engineer's approach to problem solving* [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." Koen identifies three kinds of heuristics:

1. Heuristics that lead you to take a specific action (software design patterns are one example of this kind of heuristic, but there are many other heuristics both larger and smaller than individual patterns);
2. Heuristics that shape your beliefs and values; and
3. Heuristics that guide you in the use of other heuristics (for example, pattern languages suggest paths and sequences of connected actions).[1]

Koen points out that while heuristics often contradict each other, they are still useful. Our perception of our current design context, as well as personal preferences and design values, lead us to apply particular heuristics and choose a course of action.

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; WK; Wirf20]. I've written blog posts and essays, given talks, and conducted workshops on the nature of heuristics and how to collect and share them.

My goal has been to make software designers more aware of the intrinsic value of the design heuristics that they've acquired through direct experience. I don't want them to push their personal heuristics aside in favor of expert advice. They also shouldn't take every bit of design advice they find as authoritative, but instead question whether certain advice—including a specific software design pattern—is appropriate to their design context.

I typically start out talks on heuristics by telling the story of cooking my very first Blue Apron[2] 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 point out all the places where the recipe suggests to add salt. I postulate that if I had followed Blue Apron instructions without applying any judgment, the dish would be too salty to eat.

I then share how I applied my own judgments in order to modify the instructions to fit with my understanding of what makes for a tasty but not-too-salty dish. Instead of mindlessly following their recipe (see Figure 1), I ignored several places where the recipe suggested adding salt and also made personal interpretations of other imprecise instructions (such as determining the appropriate amount of liquid for a "drizzle").

One personal heuristic I used was, "ignore instructions for adding salt if it **seems** (based on my gut feel) excessive." Also, "only add salt to taste at a step where you can actually taste its effect." Following those personal 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 about applying knowledge they've gained through previous software design experiences to new situations. Each of us have valuable insights and experiences that we should feel confident to draw upon.

---

[1] For a more detailed introduction to different kinds of heuristics see Growing Your Personal Design Heuristics Toolkit [Wirf19a].

[2] 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.

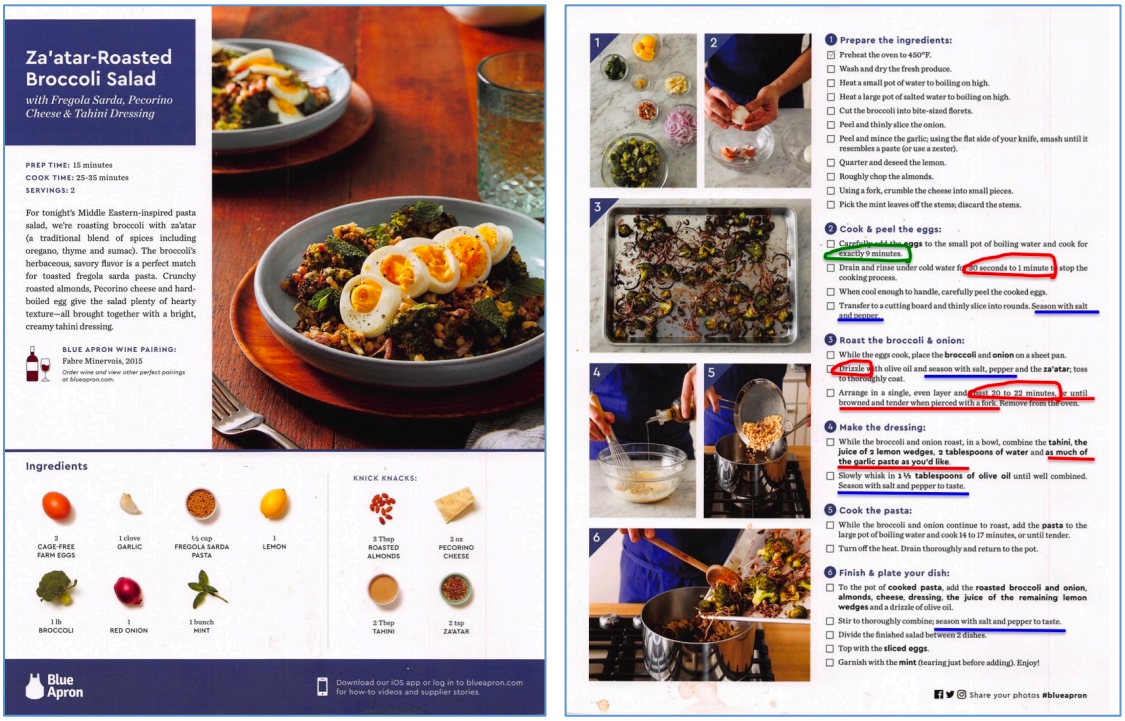

Figure 1. Blue Apron Recipe (front and back) with annotations where instructions were to add salt (underlined in blue), seemed imprecise (underlined or circled red), or overly precise (circled in green).

With my story, I was trying to illustrate the appropriateness of thinking for yourself, without needing to deeply understand the umwelt of the situation (cooking a particular dish). Simply trusting and directly following "recipes" or "patterns" because they are published or credentialed might well lead to inedible dishes or to poorly designed software.

In hindsight, I misled my audience by painting an overly simplistic picture of software design, design expertise, exercising judgment and becoming more discerning. While we should value and treasure our experiences, reflect on them, and draw upon our personal heuristics with more confidence, we should also be cautious about our heuristics' relevance to new situations. We mislead ourselves if we think we know best what to do in situations where we're not grounded with sufficient knowledge and experience. While my finished dish *looked* impressive, it tasted bland (see Figure 2).

Figure 2. The finished dish which, while looking great, lacked flavor.

3. THE BENEFITS OF KNOWING WHY THINGS WORK THE WAY THEY DO

Last December I read *How to Taste* [Sel] by Becky Selengut. It's a book about how to become a better cook by becoming more informed about taste and techniques for adjusting it. 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 am superhuman, I can't rely on instincts alone to create tasty dishes. If I follow recipes without knowing why they work, I'll merely become good at following recipes. My umwelt will be limited to being able to follow recipe instructions. I won't have learned ways to "tune" or "adapt" recipes according to personal tastes or to use the ingredients I have at hand. Furthermore, I won't know what actions to take 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 or science behind what makes a good dish good or a tasty dish tasty. 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 certain things or why steps were in a particular order. I didn't think deeply about the recipe or how to improve it, let alone why it worked. Pretty much, I followed instructions while trying to be efficient. If two steps could obviously be done at the same time, I'd do so. No need to get the water boiling before measuring out the noodles. Over time, I figured out more shortcuts. I can modify recipes in limited ways to use available ingredients or lower their fat content. My knowledge on how to do so and why certain techniques work has largely been acquired through searching the internet. And while my cooking techniques have improved, I haven't developed the ability to craft a dish with nuanced flavors, let alone improvise. I'm getting better at this, thanks to Becky Selengut's advice and several other sources that are teaching me about flavor.

Selengut 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 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 and develop discernment is to perform structured experiments after being presented with a 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. 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 during cooking. 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 Selengut'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 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 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 intended to teach me to be a better cook. In the process of preparing a dish from one of their recipes I might learn of some ingredients that I might incorporate in other dishes. But I won't know *how* to do that.

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 did learn some generally useful skills—like how to roast vegetables or mash potatoes, I didn't gain deeper knowledge that allows me to improvise or be more discerning. Their recipes focus on how to make a dish efficiently following a step-by-step process—not why it tastes the way it does or how I might alter it to suit my preferences (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 also either presented as step-by-step lists of instructions or guidelines without accompanying explanations of why it makes sense to do so or without pointers to relevant background information that might help us understand the topic more deeply. And when we join an existing development team working on some new-to-us system, we tend to go with the flow (and approach existing software artifacts without any deep understanding of the system or the values and beliefs of its initial designers). Over time we may learn these things, but this learning is mostly ad hoc and unstructured.

This is also the case when we pick up and attempt to use an isolated software design pattern. The pattern description may include a few considerations to help us determine whether it might a "reasonable" fit. It may also briefly describe a few potential consequences of applying the pattern. But patterns never come with explicit instructions on how to adapt them to specific design contexts. The descriptions are written by authors who assume the reader has the necessary background to relate this information to their particular design situation. Which is only true if the reader shares enough of a design umwelt with the pattern authors to read between the lines and enrich pattern descriptions with their own understanding.

Furthermore, in design patterns, as with recipes, 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 interchangeable behaviors. Rarely are we warned of consequences of not following a particular aspect of a pattern solution to the letter. Certain details might be important; others, not so much. It's only through direct experience that we learn these distinctions.

Chris Richardson (author of a book on Microservice Patterns [Rich] and organizer of an online community on that architecture topic), in a recent email thread with myself and Joseph Yoder, posed this question:

> *"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'?"*

I replied:

> *"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, and to have the ability to evolve microservices semi-autonomously...but still, this begs the question of what makes a microservice architecture implementation a good one.*

On the other hand, I think the patterns at the GOF level[3] 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."

My initial reaction to Chris' question was that patterns—at least the software and software architecture patterns I'm aware of—are written for 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 necessarily "vague" in the sense that they don't describe how to create such structures from scratch, nor how to modify existing systems to effectively incorporate them. Rather, they are overall structuring patterns—they aren't intended to offer recipe-like solutions.[4] I've concluded that these patterns for larger structures need to be presented differently.

Other software patterns are about much smaller design considerations (or rather coding considerations), for example how to hide implementation behind a Façade or how to define plug-replaceable behaviors using the Strategy pattern [Gamma]. When you use these 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 identified patterns or not). Usually, descriptions for these low-level patterns include sample solution sketches that appear at first glance 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 straightforwardly implementing it).

As Joe Yoder pointed 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:

"…you [Chris] 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[s], if you don't pay attention and do good modelling, you will not get the benefits and can have a worse problem…I have tried to outline some design principles for microservices. A lot of them come down to good domain modelling principles."

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

Over time and through experience we may build up a playbook of various procedures, individual design patterns and design practices. But our knowledge won't be deep, or rich, or adaptable unless salient new facts become connected to our existing knowledge and integrated into our umwelt. Even so, we may still struggle to make meaningful design 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 different fats interact with other ingredients (or going even deeper, why they have these effects and how they work at different temperatures).

Likewise, in software design, if I am a JavaScript programmer, I may learn about the latest features of the various React libraries[5] or common problems and how to overcome them.[6] But I won't really learn much about the mechanics of rendering or re-rendering web pages, or strategies for handling long lists of items.

---

[3] GOF is a reference to *Design Patterns* by Gamma et al. The four authors are sometime referred in the pattern community to as the "Gang of Four," GOF for short.

[4] Recently, Joseph Yoder and Paulo Merson have written patterns about different approaches for 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.

[5] For example, see https://brainhub.eu/library/top-react-libraries

[6] See https://brainhub.eu/library/react-js-problems

Since pattern descriptions themselves aren't complete enough for us to pick them up and use without sufficient know-how, where can we acquire that knowledge if our experience doesn't provide the necessary background?

To truly gain proficiency at software design or programming, experience alongside instruction and information that emphasizes the why along with the how is what I need.

The very first software patterns book, *Design Patterns* [Gamma], devoted the first chapter to introducing and justifying patterns. It also explained object-oriented design concepts, principles and practices, as well as the authors' design values. It introduced us to some critical "whys" behind the patterns and effective ways to design using object technology: the importance of programming to an interface; some advantages and disadvantages of object inheritance and composition; the effects of delegation on a design; and the notion that while using certain patterns increase a design's flexibility, at the same time they also add design complexity. Design always involves making tradeoffs.

This chapter was followed by a lengthy case study that illustrated how these design principles and several patterns were used in designing a document editor. This case study exposed us to important design considerations as well as insights into the authors' design umwelt. Yet how many of us bothered to read and digest this material before jumping straight to reading the patterns? I remember eagerly reading this introductory material because I wanted to compare their design values and experiences with my own.

Without such grounding and background, inexperienced designers are more likely to misinterpret these lower-level software design patterns as instructions to be followed because someone (the pattern authors or more experienced designers or an instructor) said to do so. They won't connect them with more fundamental object design principles that they are based on nor understand that these patterns are intended to be applied with discernment.

5. LEARNING TO ADAPT

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

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, turn off the oven after the timer beeps. The next level, which he calls 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 instructions, but through learned observation and perceptions, I am altering my behavior to match the current context. The third level of learning, called deutero learning goes a little meta: learning to adapt our responses based on recognizing and reacting to changing contexts. For example, with experience, I recognize that I likely need to adjust timing and temperature when using an unfamiliar oven.

Bateson points out that learning happens at various levels, regardless of our intentions (or awareness). Design habits and heuristics and values seep in to our umwelt through our experiences. And those experiences shape how we approach new learning. Deutero learning impacts the way we see things, what we anticipate, how we unconsciously behave, as well as what we perceive as "normal" responses in specific contexts. Bateson and others 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.

An experienced software designer's umwelt gives them enough design chops to know to how use a pattern in their design context, even if they may not be able to articulate what they are doing when they apply that pattern. I know that my stance is to take any description of a software pattern solution as a sketch. They are starting points. Suggestions. Heuristics to be weighed against other design options.

And if I choose to employ a particular pattern, I will always need to adapt it to fit it into my design. Invariably, my solution will be more complex and nuanced than the solution sketched out in the pattern description.

But how did I come to know this about software patterns?

Over years designing and writing software, I came to recognize that any sketch made on a whiteboard or in my design notebook wasn't going to be accurate. Nor detailed. Invariably I would uncover additional details as I implemented my designs or talked about design ideas with others. It was expected as young engineers for us to review what we were doing and why with our peers and get helpful feedback. We all did this.

Roughly during that same period, I also learned how to efficiently move between looking at detail in code and sketching out design ideas that intentionally omitted many details. If someone didn't know much about my code, I would sketch and talk about my design before we turned to look at code. Both perspectives were useful. I came to appreciate the value in shifting between these perspectives. No perspective was better or more "accurate"; it was just more or less appropriate for the context and to whom I was talking to. Thinking back, I believe that I took this same attitude into learning about and using software design patterns. I expected to learn a little about a pattern by reading it or seeing examples of it used in various situations. But should I choose to use that pattern, I expected to learn much more during the process of adapting it to my design.

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 of their ingrained design heuristics may be based upon premises that won't hold up very well under closer scrutiny. And that falling back on familiar design heuristics without considering their utility in new situations can result in unpleasant surprises. Worse yet, when we don't know some of the whys behind our personal heuristics, we may not know what appropriate alternatives to consider or corrective actions to take to get our designs back on track.

There are many situations when it is perfectly fine to act on instinct (or rather, unconscious learned behaviors and tacit 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 directed attention.

In my talks I speak about 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. 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 used for patent applications. But these notes inadvertently served another purpose—to help me recall how thoughts and design ideas progressed. This record left me some breadcrumbs that I could reflect back on.

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

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 this particular heuristic or pattern have a desired effect, and why at other times does it fail? Ask what other approaches might I consider and 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 can be little time for introspection and critique. Instead, your design efforts narrow to only making the design work well enough so that it can be put to immediate use without breaking existing functionality.

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 play a big part in growing design skills. As does practice and reflection. 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 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*

[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.

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."

There are ways to incorporate active 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, 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 design playfulness and experimentation. We should make more space for these kinds of activities. But even these efforts might not go far enough.

7. LETTING GO OF CERTAINTY

I no longer believe that salt should be added only when I can immediately taste its effect (it's easier if I can do so, but not necessary—if I trust the recipe and know why salt needs to be added, how much to add, and when to do so). But I can still slip up and make mistakes. Like adding too much salt. The recipe might've been unclear, I might've misunderstood some advice, or perhaps the advice was misguided.

Consequently, I am now learning ways to fix oversalted dishes. If fortunate, I simply can add more ingredients to cut the saltiness. That's easiest, but not always practical. I might add something acidic like lemon juice or vinegar, or something sweet, or something buttery or creamy like avocado. It all depends on where I want to take the flavor profile[8] of my dish—assuming I know where I want to take it.

And yet, I'm still not expert at this. I'm uncertain that any new-to-me salt reducing heuristic will have the desired effect or please my palate. So, while I approach saltiness-reducing efforts armed with others' advice, I remain curious and questioning. Why did a particular tactic seem to work? How much of a saltiness-reducing ingredient should I add?

As I come to understand the utility of new heuristics, I find that I may also need to let go of some of my strongly held beliefs or perceptions (or perhaps, recalibrate their applicability).

As I've relaxed my strong preference for solving most software design problems using objects, I've come to appreciate non-object-oriented design approaches. I'm still most comfortable fashioning solutions using objects and well-known software patterns. But I recognize there are many contexts where other design paradigms and patterns shine. It's not that I've abandoned my cherished heuristics so much as I've sorted through and rearranged my umwelt. I've situated new-to-me and useful heuristics and approaches alongside older, cherished ones. Both my merkwelt (perceptions) and wirkwelt (actions) have been enriched by stepping outside my comfort zone and trying out new-to-me ways of designing software.

In doing so, I've found that sometimes I can directly transfer some of my existing knowledge to new design paradigms. For example, the Strategy pattern can be just as easily implemented as a set of functions—as long as there is a single behavior associated with the strategy and no need to retain any state. And the Decorator pattern can be implemented as composable functions.

At other times, I can't see how to integrate new heuristics into my umwelt. Before these heuristics become useful, I need to sort them out and appropriately arrange them into my umwelt. When I don't see an easy fit, I lean more on experts' advice on how best to make use of them. Since I'm not skilled at using new heuristics, they may feel awkward or unnatural. I may need more practice. Perhaps, with practice, I'll find that they aren't a good fit. If so, I set them aside.

It is only when I temper others' advice with what I learn through direct experience, practice and reflection, does this knowledge feel like it is truly becoming part of my umwelt.

Whether novice or expert, breaking out of design "habits" or growing new skills as Woody Zuill's story illustrates—and my experiences corroborate—need not be painful. As you actively expand and tune your design umwelt you may feel uncertain. Some new-to-you heuristics require more practice and consideration than others before they make sense. Regardless, most heuristics will need to be adapted to fit your design context. And you will be better to equipped to do this kind of tinkering if you learn some things about why these heuristics work the way they do as you try them out.

---

[8] After devouring *How to Taste,* I purchased *The Flavor Matrix: The art and science of pairing common ingredients to create extraordinary dishes* by James Briscione. This book is beyond my current expertise. And yet I'm hopeful that when I dig into it with a little more experience, a sense of wonder, and an experimental attitude, that my flavor-creation skills will grow.

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 who my intended audience is and what's in it for them. I'd also like to thank my virtual writers' workshop colleagues for their candid reactions and encouragement. Also, thanks for pointing out places where you thought I needed to say more. Thanks to Jordan Wirfs-Brock for her fresh insights and HCI design perspective. And finally, thanks to Allen Wirfs-Brock and Lise Hvatum, two of my most supportive and consistently thoughtful critics.

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

[Ux] Uexküll, J. von. (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/