Should we stop writing design patterns?

Rebecca Wirfs-Brock, Wirfs-Brock Associates

This essay reflects on experiences I've had and what I've learned over the past 15 years of writing software design patterns. I've come to believe that if we want to broadly increase pattern literacy, relevancy, and long-term impact, some things need to change. Most importantly, I believe that change needs to start with pattern writers, like me. Instead of indiscriminately writing even more patterns I should focus more on connecting, relating, promoting, and refreshing existing impactful patterns. Changes also need to be made in how our community of long-time pattern authors and advocates present, organize, and promote patterns to the rest of the world.

Design Patterns, Process Patterns, and Learning from Writing Patterns

I've written dozens of patterns with several colleagues and friends. The topics range from software requirements to software design and architecture to software development process and practice patterns. Some of our patterns have been technical, detailed patterns for a specific architecture style (specifically, Adaptive Object Models) [AHLSWY, WWY07, WYW08, WYW09, HNSWY10, HLNSWY10].

Others have been about architecture practices on Agile projects [WY, WYG]. Still others have been about patterns for improving software quality [YWA2014, YWW2014a, YWW2014b, YWW2015, YWW2016a, YWW2016b]. I've even ventured to write patterns about managing and evolving meaningful product backlogs for complex, long-lived engineering products [Hva2015, Wirf2016, Hva2017, Hva2018, Wirf2019].

The discipline of writing patterns has been mostly fun and only occasionally challenging. Surprisingly, one of the most difficult aspects of writing patterns has been getting sufficiently rich and useful critiques from pattern writing workshops. Sometimes my fellow writing workshop colleagues have been helpful, at other times they've barely grasped my patterns. This is mostly because PLoP conferences have become venues for reviewing many more kinds of patterns than those for designing, building, and managing software development products and projects. Not many experienced designers and architects attend pattern writing conferences these days. So these days, most of my exposure to new design ideas and inspiration comes from outside the patterns community.

Through my pattern writing, I've immersed myself in pattern trivia and become a student of patterns and Christopher Alexander's writings and philosophy. And yet, I still feel like an patterns community outsider. *Inside* the patterns community I may be perceived as somewhat of a pattern geek; *outside* this community I talk with many other developers and designers about their practices and techniques, and design guidelines and heuristics and successes and failures. Only occasionally do I talk to them about patterns. There are benefits to being an outsider. I feel an outsider to any community I am part of, by the way. This feeling isn't unique to patterns. The fact that I don't feel that I am singularly defined by any particular community allows me to move between communities and spread and learn new ideas.

This means that I'm not defined by pattern writing. Nor am I defined by Domain Driven Design, Agile development, architecture, or Open Space communities. What I do care most about, and this transcends communities, is about learning and sharing expertise and growing awareness in and appreciation in others about how to sustainably design and build useful software systems.

None of the patterns I have written have had much, if any, impact on this larger software development community.

Could it be that we in the patterns community are operating at a meta level of software development while most developers are interested in more concrete problems and more specific, detailed advice?

Or perhaps it was a matter of packaging. Some patterns we wrote were part of patterns collections. But they were scattered over a series of papers published over several years in different PLoP proceedings. This makes it difficult for all but the most dedicated reader to find. For various reasons we have not yet published these patterns in books which perhaps would have made our patterns more accesible.

I suspect that other factors contributed to our patterns' obscurity as well—a matter of poor timing, lack of approachability of written pattern forms to my targeted audience, limited applicability, lack of promotion, or lack of any deep and lasting connections between our patterns and other design practices and patterns and schools of thought. My object design books [Wirf90, Wirf02] have had a far greater impact on designers. This is part due to the readability/approachability of the books, but also partly due to timing. Although I think my second book on object design had even more valuable contributions to design thinking, it is the first book written in 1990 that was (and still is) more widely recognized and noticed.

Regardless, writing these patterns has had a tremendous impact on me.

Some lessons I've learned:

Many existing software design and architect patterns are of interest only to those working in a narrow software niche.

Toiling away writing about a variant of a particular pattern may add to our overall body of knowledge (yes, I wrote a pattern named *Adaptive Object Model Builder*

[WYW09], a variant of the *Builder* pattern, and several Adaptive Object Model rendering patterns, but this effort was of limited value. Such knowledge (especially since it is extremely narrow) can be of value only if that knowledge can be readily shared and made accessible to others working in that same software design space. Not many are building adaptive object-model architectures.

In hindsight, I caution potential authors who are working in a specialized software field, to be aware—the effort you take to document your work as patterns may not have any discernible effect to advancing your field. Pattern writing can help you understand and develop your contribution. You'll learn how to express the design constraints and forces that are balanced by a particular pattern. You'll learn how to create and illustrate exemplary solutions. But the darker truth is that publishing your knowledge only in pattern form most likely limits the exposure of your work to only those few who are engaged in writing patterns community.

Many software pattern descriptions need refreshing.

This is especially true of many early software design "proto-patterns." Christopher Alexander and colleagues in the preface to *A Pattern Language* observed: "...each pattern represents our current best guess as to what arrangement of the physical environment will work to solve the problem presented. The empirical questions center on the problem—does it occur and is it felt in the way we describe it?—and the solution—does the arrangement we propose solve the problem? And the asterisks represent our degree of faith in these hypotheses. But of course, no matter what the asterisks say, the patterns are still hypotheses, all 253 of them—and are, therefore, all tentative, all free to evolve under the impact of new experience and observation."

Alexander expected their patterns to evolve under new situations and design contexts. What his expectations for how these new insights could be conveyed to other designers and architects and builders isn't so clear. In later works Alexander speaks of creating project-specific pattern languages (adding to and emphasizing partiular patterns) that would provide high level guidance for a specific architecture project. Project languages consist of unique and customized sets of patterns, appropriate to a specific context.

We software designers and architects have the challenge—or rather an opportunity—to illustrate and share our design insights more readily.

Even though many early patterns authors have not updated their well-known pattern descriptions, others with unclear connections to the original authors or the PloP comunity, are admirably taking up this effort.

For one good example, John Thompson offers web pages that give an updated presentation of the 23 patterns in *Design Patterns* cast in terms of the Java Spring Framework [Thom]. These descriptions are well-motivated and provide a quite approachable introduction for Java programmers. The *Observer* pattern description has been motivated by a more modern application (registering and receiving tweets from those you follow). And the example solution uses Java interfaces instead of the original class-based solutions to define Obeserver and Subject behaviors. And to make the pattern even more relevant, a specific example of how this pattern is applied in the Spring Framework is illustrated.

The author also writes a very good discussion of the controversial *Singleton* pattern, demonstrating how to create a threadsafe version of a Singleton and offering strongly worded opinions on where it *might* be appropriately and why it should be sparingly used.

Another example is Brandon Rhodes' Python patterns guide website [Rhod]. In addition to showing examples of each GOF pattern, he also presents additional common Python patterns and discusses at length how by using the deisgn principles in *Design Patterns* one can come to a robust, comprehensive and flexible design solution. For example, in the discussion of the principle, favor composition over inheritance, different design solutions to logging are shown and discussed. These range from *Adapter* to *Bridge* and *Decorator* pattern implementations. Then, in a section titled "Going beyond the Gang of Four patterns" he illustrates how the Python's logging capabilities implemented in the Standard Library implemented even more flexibility: not only supporting multiple filters, but multiple output streams for log messages.

These two sites provide quite useful information for the design curious Java or Python programmers at the level of detail that programmers can relate to, with plenty of code along with thoughtful design discussion and commentary. [1]

Many other patterns are broadly useful, even if they are not well known.

A notable example of useful patterns that have slipped into relative obscurity are the software re-engineering patterns described in *Object-Oriented Software Reengineering Patterns* by Serge Demeyer, Stéphane Ducasse, and Oscar Nierstrasz [Dem]. Recently the authors have reclaimed the copyright to this book and now have made an online version freely available. [2]

[1] It is interesting to note that the authors of these refreshed *Design Patterns* only show code solutions; none of their patterns are illustrated with UML class or sequence diagrams.
[2] The current packaging of these patterns is now available as an Open Textbook Library text. See https://open.umn.edu/opentextbooks/textbooks/object-oriented-

[2] The current packaging of these patterns is now available as an Open Textbook Library text. See https://open.umn.edu/opentextbooks/textbooks/object-oriented-reengineering-patterns with access with attribution licensing.

Each chapter starts with a pattern map illustrating potential sequences through the patterns the chapter based on actions (for example, see Figure 1 for the pattern map for Chapter 4).

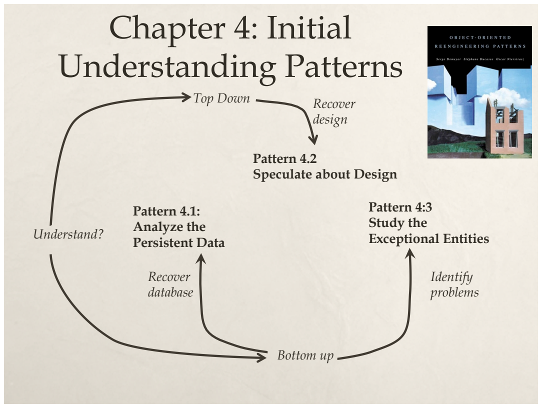
Chapter 4: Initial Understanding Patterns
Top Down
Recover design
Pattern 4.2 Speculate about Design
Pattern 4.1: Analyze the Persistent Data
Pattern 4:3 Study the Exceptional Entities
Understand?
Recover database
Identify problems
Bottom up

Figure 1. Each chapter in *Object-Oriented Reengineering Patterns* is a small language

Many of these patterns are still relevant in today's development context (although they too, could benefit from some updates. For example, one of the patterns is *Do a Mock Installation*. These days, the build process, even for a legacy system, typically has been at least partially automated; consequently an appropriate substitute for this might be a pattern named *Build the System and Map Dependencies*). In a previous essay I observed [Wirf] that, "Unintentionally, the biggest misstep these authors made was titling their book, *Object-Oriented Reengineering Patterns.*" Object technology patterns are only mentioned in the last two chapters, and some of these object-technology specific patterns could easily be rewritten to be more generally applicable in both functional and object programming implementations: *Move Behavior Close to Data, Eliminate Navigation Code, Factor out State, and Factor out Strategy*. Perhaps if retitled *Software Reengineering Patterns* (and new additional chapters described several functional programming language implementation patterns) these patterns could be rescued from obscurity.

Another example of a handful of useful but unknown patterns comes from my own work. In a 2008 pattern paper [WY] I wrote with Joe Yoder, descriptions of three patterns for sustaining software architecture. Joe, among other accomplishments, is known as one of the co-authors of the *Big Ball of Mud* pattern [Foot], arguably one of the most misunderstood patterns of all time.[3] *Paving Over The Wagon Trail* is a pattern for building a tool (most likely to generate code) that allows repetitive, error

[3] People often refer to the Big Ball of Mud as an anti-pattern, an architecture that is sprawling and unmaintainable, that you should try to re-work. Instead, the Big Ball of Mud paper contains a number of smaller practical patterns for containing and managing complexity in a sprawling legacy system in addition to its heart-felt and humorous discussion of how it is better to live with mud (in appropriate places of a system) rather than to try to fix things up and make a totally new, clean architecture.

prone programming tasks to be eliminated. *Wiping Your Feet at the Door* is a pattern for cleaning up data/transforming it at the "edges" of a system in order to reduce internal complexity. The third pattern was a well-intentioned effort to make programming simpler that instead reinforced poor programming practices and exacerbated the erosion of the architecture, *Paving over the Cowpath*. In this pattern we took care to explain why this is not an anti-pattern, but instead a darker form of well intentioned tool tinkering. These patterns help with both prevention of design decay and sustaining complex, evolving architectures.

Arguably these patterns are relevant today, but as they never gained much visibility they are at an even higher risk of fading into obscurity. Perhaps I am humoring myself thinking that they haven't already disappeared from sight. I suspect there are several reasons for their obscurity. We were far too clever and culturally limiting with their naming. The *Big Ball of Mud* after all, inspired us, to give them names that implied muddy pathways, and trails, and the need to clean data at system entryways. This detracted from our patterns' potential "stickiness." Perhaps a more important contribution to their obscurity, I suspect, was that we did not actively connect them to other well-known related patterns such as Eric Evan's *AntiCorruption Layer* patterns [Evan] or Big Ball of Mud Patterns. We certainly knew about these patterns, but we were more interested in writing about new patterns which incorporated our shared knowledge and experience than in hooking them to the broader body of existing patterns and promoting their recognition and use.[4]

Patterns, no matter how useful, cannot not survive in isolation. Imagine having just read APL's *Window Place* or *Light on Two Sides* pattern with out having a larger context of designing a dwelling as described in the pattern a *House for a Small Family* or a *Building Complex.* Not only should patterns be connected; there naturally are larger "enclosing" patterns that should be identified which provide a home for and structure for patterns with a smaller scope.

To stay relevant, good software design patterns need to be connected to other, related patterns and heuristics and practices (regardless of who authored them). And they shouldn't be cast as too clever or culturally quaint.

Many software design and architecture pattern descriptions, are overly specific and imply prescriptive technology solutions.

It is easy to criticize the original *Design Patterns* descriptions as of 2020 as being outdated. But this isn't a fair critique. At the time early pattern authors wrote their patterns, they described what they knew and directly experienced. No speculation or innovation or generalization; design patterns described design phenomena

[4] Perhaps this urge to create rather than integrate shouldn't be surprising, but neither the *Big Ball of Mud* nor Domain Driven Design patterns have been updated by their original authors. Fortunately, other thought leaders in the DDD community have written several books popularizing several complementary Domain Driven Design practices and patterns.

observed in multiple, pre-existing, successful software systems. *Design Patterns* was written when object-technology was gaining prominance and object design solutions were common (they still are, but are not as predominant).

Even the Spring Framework design patterns author gives a somewhat limited view of these patterns' utility as evidenced by their introduction to his work: "The GoF wrote the book in a C++ context but it still remains very relevant to Java programming. C++ and Java are both object-oriented languages. The GoF authors, through their experience in coding large-scale enterprise systems using C++, saw common patterns emerge. These design patterns are not unique to C++. The design patterns can be applied in any object oriented language."[5]

However, there is nothing object technolgoy specific about an *Adapter* or a *Facade* or an *Observer* even though *Design Patterns* was explicitly written as a collection of Object Design solution patterns illustrated with pre-UML class diagrams and C++ code snippets. Even today, most developers construe the original 23 *Design Patterns* as patterns as relevant only to object-oriented software solutions.

However, it is straightforward for me to transpose these patterns to different contexts. I know I can employ an *Adapter* or *Bridge* or *Strategy* regardless of technology. A more inexperienced designer does not make these connections so easily. As someone who has designed software for decades, I can see and subsequently abstract pattern solutions in order to re-apply them to new technologies even though the original pattern descriptions might have been overly constrained or deceptively simplified.

It is also interesting to note that the Spring Java refreshed versions of *Design Patterns* include examples written in Java, along with textual descriptions. There are no Class diagram illustrations or more abstract depictions of pattern solutions. These days, even a more abstract representation of a design solution—say a UML class or sequence diagram fragment—is becoming increasingly rare because such representations are unknowable and incomprehensible to most programmers.

The fact that I've had direct experience making as well as seeing pattern solutions implemented in several different programming languages gives me a perspective that is lacking in someone with only Java programming experience. Experienced Java developers who have seen software patterns implemented only in Java only know them in that context. Paradoxically, this limits their design reach; while at the same time strengthens their Java programming efficacy.

[5] In fact, the authors of *Design Patterns* were familiar with C++ and Smalltalk. But as Smalltalk's popularity was fading at the time the book was written, they made the decision to use one programming language and they chose C++.

There's an unresolvable tension between presenting easily understood concrete solutions and more generalizable abstractions

However useful software design patterns may be, unless they have been refreshed and contextualized for today's technologies and software designers, they will be difficult for newcomers to grasp. An accessible introduction to a specific software pattern and its significance ideally provides a context and a concrete example that can be readily latched onto. When the technology changes—to stay approachable—that solution will most likely need updating.

Paradoxically, it is the concreteness of a pattern solution that deceives us into believing that "what we see is all there is" [Kahn]. As Rudolph Arnheim in *Visual Thinking* [Arn] observes, "The more perfect our means of direct experience, the more easily we are caught by the dangerous illusion that perceiving is tantamount to knowing and understanding."

To present the essence of a pattern, however, requires a different angle. That form needs a sparser description with a simple, exemplary sketch of the problem and solution. It needs to convey its significance and at the same time enable us to adapt it to our specific design situation. Equally important, that pattern needs to be located among other patterns and within a larger design context.

Both kinds of descriptions are valuable—just to different audiences.

To date we haven't identified ways to "label" our patterns descriptions with cautionary advice. So we pattern authors need to be careful as we choose to represent solutions to take care to explain our choices to our readers.

Back to my Patterns Beginnings

Looking back to 2006, I wrote my first patterns with Paul Taylor and James Noble [WTN]. We explored writing patterns for conceptualizing problems rather than designing solutions to those problems. We were eager to make connections (and create or link a rich network of patterns) that spanned requirements *and* design.

There are times when software designers don't see clearly the what the problem is. In our paper we asked, "What if we find ourselves washing around in the amorphous problem space, unable to get a foothold on anything to bear the weight of a [design] pattern or to anchor a fragment of architecture? Is there another kind of pattern that helps to locate our thinking early in the analysis and conceptualization of systems and solutions? Do patterns in the problem space exist?"

We used Michael Jackson's *Problem Frames* [Jack] as a basis for this pattern writing experiment. Jackson's problem frames are intriguing because they build on a recognition of generic problem types, based on structures and relationships between domains and designed system elements (Jackson calls these "machines"). Problem frames are based on the philosophy of phenomenology, which firmly places us in a world of concepts, domains, phenomena and machines—in our case as software desginers those machines are software mechanisms of our own design—which interact with the elements of the problem's enveloping context in order to have a desired effect upon the world.

Jackson described five different problem frames: Required Behavior, Commanded Behavior, Information Display, Simple Workpieces, and Transformation.

Let me briefly characterize each frame. A Required behavior frame deals with a class of problems where you want to control state changes of some *thing* outside the boundaries of your software machinery. A Commanded Behavior problem frame is about controlling changes to some *thing* based on either an operator or user's commands. An information frame is about problems where there is a need to produce information about observable phenomena (usually over time). A Simple Workpieces Frame addresses the problem of creating tooling, which enables users to create and manipulate structures. Finally, a Transformation frame is about problems of converting input to one or more outputs.

Jackson illustrated each problem frame with a schematic drawing and discussed their specific concerns.

I recall that when firsting read Jackson's work some years prior to my pattern writing experiment, that I hoped additional frames would soon be added by an active problem framing community.[6] The problem frames Jackson described, or at least the way they were presented in his book, didn't seem immediately relevant to the problems I frequently encountered in IT and software engineering. Jackson's frames seemed most appropriate for characterizing requirements of physical control systems. However, I found that with a little bit of mental effort that I could "stretch" his conceptual framework to fit into the IT and engineering problems I was designing solutions for.

For example, instead of controlling a device, I might design software that was "controlling" the behavior of an external software system that I couldn't directly probe for whether it had acted on my software's requests. I was modernizing and refreshing Jackson's frames to better fit my design context. But I could only stretch his frames so far and they only covered so much of my problem territory.

[6] I remember when I first reviewed the *Design Patterns* book [Gamm] that I hoped for more patterns, too. In fact, in my review of the draft of the book to Addison-Wesley I suggested that they publish the book in a form where installments and additions could be made on a regular basis. This notion was similar in concept to the yearly addendum to the Encyclopedia Britannica, which could be purchased annually to keep the encyclopedia up to date.

Once I conceptually understood problem frames, I caught glimpses of them everywhere. Complex software systems and systems interacting with other systems and databases tend to have multiple overlapping frames. So after I had appropriately framed a situation, there were salient questions I could ask to uncover more information about the nature of the problem at hand. For example, here are some questions to ask about required behavior problems:

- What external state must be controlled?
- How does my software find out whether its actions have had the intended effect? Does it need to know for certain, or can it just react later (when the state of some thing is not as expected)?
- What should happen when things get "out of synch" between the software system and the thing it is supposedly controlling?
- How and when does my software decide what actions to initiate?
- Is there a sequence to these actions? Do they depend on each other?
- Are there complex interactions with my software and the thing under its control? Should there be?
- Can I view the connection between my software and the thing under control as being direct (easier) or do I need to consider that it is connected to something that transmits requests to the thing being controlled (and that this connection can cause quirky, interesting behavior)? If so, then I may need to understand more about the properties of this "connection domain" that stands between my software and the thing being controlled.

In our problem frame patterns paper we remarked that, "[t]he fact that we put problem frames into pattern form demonstrates that when people write specifications, they are designing too—they are designing the overall system, not its internal structure." And while problem frames are firmly rooted in the problem space, to us they also suggested potential pattern solution spaces to explore. For example when solving translation problems it seems reasonable to check out software design patterns about how to write parsers, or to consider the *Command* pattern when designing a solution to a Commanded Behavior problem (or most frames involving a user-operator domain). And Required Behavior problems suggest investigating event and event handling patterns, finite state machines, or reactive system design patterns.

These are fairly straightforward connections for *me* as an experienced software designer to make. But this is because I know of many software design and architecture patterns as well as other design techniques, practices, architecture styles, and design heuristics. Problem framing is simply one, among many ways, to explore a problem space—a conceptual tool I use to focus as I untangle complex system requirements.

At that time I learned about problem frames, there were other better-known analytic techniques available. We analysts and designers were busy writing Use Cases, creating context diagrams, workflow, data and object diagrams. I integrated framing into my existing bag of tricks for understanding the problem space and quietly moved on. Problem framing didn't replace any analysis technique I already knew; it just slipped in amongst them all as an imperfect backdrop.

So, did problem framing help *me* be a better designer? I'm not sure.

The path from framing a problem to choosing an appropriate software architecture or set of design patterns is roundabout at best. We remarked in our paper that, "Patterns work like a ladder in the 'Snakes and Ladders' board game—given a known context and problem (square on the board) they give us a leg-up to a higher place. Design patterns fall squarely in the middle of the solution space and provide object-oriented fragments[7] of structure to resolve solution space forces."

While software design patterns are about designing things, there are also patterns in the problem space, too. Once you "see" them, applying the lens of a particular frame leads you to ask focused questions about a software system's requirements. But that's about it. The answers to these questions don't directly lead you to specific design patterns and approaches; they simply raise your awareness of what might be the harder problems to solve.

Did problem framing help others be better analysts? In general, I'd say no.

I found teaching others about problem frames to be an abject failure. Problem frames confused my students. After a few failed attempts at trying to get them to appreciate problem frames in all their gory detail—how to identify them, draw them, and describe their concerns, I dropped the idea of explicitly teaching them entirely. They didn't see the point.

They did, however, find it useful to have sets of related questions they could ask in order to gain further insight into their system's requirements. That those questions were related to the specific concerns relevant to each problem frame was a distraction. But, shh!! No need to tell my students about problem frames.

Learning the mechanics of writing clearly and at the appropriate level of detail (and some questions they might ask to get at those details) were practical skills that they could absorb and appreciate. The conceptual backdrop of problem framing was unnecessary.

[7] To put this in historical context, we were awash in object technology, design, and architecture patterns as of 2006: *Design Patterns* had been published with great success followed by Fowler's *Analysis Patterns* (1996), *Patterns of Enterprise Application Architecture* and *Object-oriented Reengineering Patterns* in (2002), *Domain Driven Design* (2003), and three of the five volumes of *Pattern-Oriented Software Architecture*.

Framing is simply one way to gain some design perspective—regardless of whether I can directly employ those insights into patterned design solutions.

Some Conclusions and a Call to Action

As designers and architects we would like it to be straightforward to link problems to potential solutions. But I find it takes a lot of hard work, experimentation, and thinking before I procede to design with confidence. And despite knowing many, many patterns, I recognize they too only offer general outlines of potential approaches. I still have to fill in many details and make decisions about my design unaided by patterns. And yet, I still have faith in software design patterns and wish they were more readily available and more useful.

Over time, the existing body of software design patterns has become sprawling, disorganized, outdated, and unknown or undervalued by many software developers. If we as a patterns community want to promote software design pattern literacy, relevancy, and long-term impact, something needs to change.

To broaden our efforts perhaps we should perform some bold experiments:

Newly published patterns are only infrequently located and anchored to the existing software pattern landscape. Consequently, there is little coherence to the large body of existing software patterns. Maybe the existing body of software patterns is too sprawling to put back together in any coherent way. But we won't know until we try.

Instead of only holding patterns conferences whose focus is to encourage new authors and students to write yet more patterns; could we turn our attention toward organizing, updating, re-mining, promoting, connecting and re-connecting, and rescuing the existing body of existing patterns?

Perhaps now, we could start a virtual effort to organize and discuss and promote patterns. We in the patterns community have a unique perspective that could help weave together some disjoint software patterns as well as consolidate the "core," more timeless design patterns from which other designers can draw upon. We have a golden opportunity to share and recommend what we find to be the most important patterns and practices for creating sustainable software.

We could create also create a forum for designers to share their insights into the many detailed decisions they encounter as they implement these patterns and the gaps they see in our patterned knowledge.

Maybe instead of only having pattern writing conferences we need to start holding virtual software pattern mining, design heuristic hunting, and refining events. And perhaps, we should start pubishing patterns differently. We could designate one or more commentators to write an accompanying precis as well an analysis that connects these newly written patterns to the pre-existing body of knowledge. And we could even be so bold as to speculate on those new patterns' significance and utility.

So, is it time we stop writing patterns? Of course, not. Not entirely.

But I think it is an opportune time for me to take a step back from cranking out lots of pattern and give myself some space and time to write, reflect, and work on the preservation, restoration, and promotion of those more important patterns that already have been identified and to identify gaps that need filling. Will you join me in this effort?

References

[AHLSWY] Acherkan, E., Hen-Tov, A., Lorenz, D., Schachter, L., Wirfs-Brock, R., and Yoder, J. "Dynamic Hook Points." Proceedings of AsianPLoP 2011
[Arn] Arnheim, R. Visual Thinking, University of California Press; Second Edition, Thirty-Fifth Anniversary Printing, 2004
[Dem] Demeyer, S.,Ducasse, S., Nierstrasz, O. *Object-oriented Reengineering Patterns*, Morgan Kaufman, 2003.
[Evan] Evans, E. *Domain-Driven Design: Tackling Complexity in the Heart of Software,* Addison-Wesley, 2003.
[Foot] Foote, B., Yoder, J. "Big Ball of Mud" in Proceedings of the Fourth Conference on Patterns Languages of Programs (PLoP '97/EuroPLoP '97) Monticello, Illinois, 1997. Also in *Pattern Languages of Programs Design 4*, edited by Neil Harrison, Brian Foote, and Hans Rohnert. Addison-Wesley, 2000.
[Gamma] Gamma, E., Helm, R., Johnson, R., Vlissides, J. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.
[HNSWY10] Hen-Tov, A., Nikolaev, L., Schachter, L., Wirfs-Brock, R., and Yoder, J. "Adaptive Object-Model Evolution Patterns." Proceedings of the 8th Latin America Conference on Pattern Languages of Programs (SugarLoaf PLoP 2010), 2010.
[HLNSWY10] Hen-Tov, Lorenz, D., A., Nikolaev, L., Schachter, L., Wirfs-Brock, R., and Yoder, J. "Dynamic Model Evolution." Proceedings of the 17th Pattern Language of Programing Conference (PLoP 2010), 2010.
[Hva2015] Hvatum, L. and Wirfs-Brock, R. "Patterns to Build the Magic Backlog". 20th European Conference on Pattern Languages of Programming (EuroPLoP), EuroPLoP 2015, 2015.
[Hva2017] Hvatum, L. and Wirfs-Brock, R. "Pattern Stories and Sequences for the Backlog: Expanding the Magic Backlog Patterns". 24th conference on Pattern Languages of Programming (PLoP). PLoP 2017, 2017.
[Hva2018] Hvatum, L. and Wirfs-Brock, R. "Program Backlog Patterns: Applying the Magic Backlog Patterns". 23rd European Conference on Pattern Languages of Programming (EuroPLoP). EuroPLoP 2018, 2018.
[Jack] Jackson, M. *Problem Frames: Analyzing and structuring software development problems*, Addison-Wesley, 2001.
[Rhod] Rhodes, B. Python Design Patterns. Website. April 2020. https://python-patterns.guide.

[Thom] Thompson, J. Web page. Gang of Four Design Patterns. April 2020. *https://springframework.guru/gang-of-four-design-patterns/*
[Wirf] "Are Software Patterns Simply a Handy Way to Package Design Heuristics?"
[Wirf90] Wirfs-Brock, R., Wilkerson, B., Wiener, L. *Designing Object-Oriented Software.* Prentice Hall, 1990.
[Wirf02] Wirfs-Brock, R., McKean, A. *Object-Oriented Design: Roles, Responsibilities, and Collaborations*. Addison-Wesley, 2002.
[Wir2016] Wirfs-Brock, R. and Hvatum, L. 2016. More Patterns for the Magic Backlog. 23rd Conference on Pattern Languages of Programming (PLoP 2016), 2016.
[Wirf2019] Wirfs-Brock, R. and Hvatum, L. 2019. "Who Will Read My Patterns? On Designing a Patterns Book for Targeted Readers". 26th Conference on Pattern Languages of Programming (PLoP 2019), 2019.
[WTN] Wirfs-Brock, R., Taylor, P., Noble, J. "Problem Frame Patterns: An Exploration of Patterns in the Problem Space" in Proceedings of the 13th Pattern Language of Programs Conference (PLoP 2006), 2006.
[WY] Wirfs-Brock, R., Yoder, J. "Patterns for Sustainable Architectures" in Proceedings of the 19th Pattern Languages of Programs Conference (PLoP2012), 2012.
[WYG] Wirfs-Brock, R., Yoder, J., Guerra, E. "Patterns to Develop and Evolve Architecture During an Agile Software Project." Proceedings of the 22nd Pattern Languages of Programs Conference (PLoP 2015), 2015
[WYW07] Welicki,L, Yoder, J., Wirfs-Brock, R. "Rendering Patterns for Adaptive Object Models." Proceedings of the 14th Pattern Language of Programs Conference (PLoP2007), 2007.
[WYW08] Welicki,L., Yoder, J., Wirfs-Brock, R. "The Dynamic Factory Pattern" Proceedings of the 16th Pattern Language of Programs Conference (PLoP 2008), 2008.
[WYW09] Welicki,L., Yoder, J., Wirfs-Brock, R. "Adaptive Object-Model Builder." Proceedings of the 16th Pattern Language of Programs Conference (PLoP 2009), 2009.
[YWA2014]Yoder J., Wirfs-Brock R., and Aguilar A., "QA to AQ: Patterns about transitioning from Quality Assurance to Agile Quality," 3rd Asian Conference on Patterns of Programming Languages (AsianPLoP), 2014.
[YW2014a]Yoder J. and Wirfs-Brock R., "QA to AQ Part Two: Shifting from Quality Assurance to Agile Quality," 21stConference on Patterns of Programming Language (PLoP 2014), 2014.
[YWW2014b]Yoder J., Wirfs-Brock R. and Washizaki H., "QA to AQ Part Three: Shifting from Quality Assurance to Agile Quality: Tearing Down the Walls," 10thLatin American Conference on Patterns of Programming Language (SugarLoafPLoP 2014), 2014.
[YWW2015]Yoder J., Wirfs-Brock R. and Washizaki H., "QA to AQ Part Four: Shifting from Quality Assurance to Agile Quality: Prioritizing Qualities and Making them Visible." 22nd Conference on Patterns of Programming Language (PLoP 2015), 2015.

[YWW2016a]Yoder J., Wirfs-Brock R. and Washizaki H., “QA to AQ Part Five: Being Agile at Quality: Growing Quality Awareness and Expertise.” 5th Asian Conference on Patterns of Programming Language (AsianPLoP 2016), 2016.
[YWW2016b]Yoder J., Wirfs-Brock R. and Washizaki H., “QA to AQ Part Six: Being Agile at Quality: Enabling and Infusing Quality,” 24th Conference on Programming Patterns of Programming Language (PLoP 2016), 2016.