Should we stop writing design patterns?

REBECCA WIRFS-BROCK, Wirfs-Brock Associates

This essay reflects on my experiences of the past 15 years writing software design patterns and posits that if the patterns community wants to broadly increase pattern literacy, relevancy, and long-term impact, some things need to change. Those changes need to start with pattern writers—like me. Instead of indiscriminately writing ever more patterns, we should focus more on connecting, relating, and refreshing the abundance of existing impactful patterns. Changes also need to be made in how we the community of long-time pattern authors and advocates present and promote patterns to the rest of the world.

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

ACM Reference Format:

Wirfs-Brock, R. 27th Conference on Pattern Languages of Programming (PLoP), PLoP 2020, Oct 12 -16, 2020, 11 pages.

1. INTRODUCTION

This essay reflects on what I've experienced and learned over the past 15 years of writing software design patterns. I've come to believe that if the software patterns community wants to broadly increase pattern literacy, relevancy, and long-term impact, some things need to change. Those changes need to start with pattern writers—like me. Instead of indiscriminately writing ever more patterns I should focus my attention on connecting, relating, and refreshing the abundance of existing impactful patterns. Changes also need to be made in how our community of long-time pattern authors and advocates present and promote patterns to the rest of the world.

I've written dozens of patterns with 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]. Some have been about architecture practices on Agile projects [WY, WYG]. Others have been about improving software quality [YWA2014, YWW2014a, YWW2014b, YWW2015, YWW2016a, YWW2016b]. We've also written patterns about managing and evolving product backlogs for complex, long-lived engineering products [HW15, WH16, HW17, HW18, WH18, WH19].

The discipline of writing patterns has mostly been fun and 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 our patterns as their expertise lies elsewhere. PLoP (Pattern Languages of Programs) conferences have become venues for reviewing many different kinds of patterns. Not many experienced software designers and architects attend pattern conferences these days. Consequently, those who are unfamiliar with designing, building, and managing software development products and projects often are among those who critique my patterns. And perhaps, not unsurprising, most of my exposure to new ideas and software design 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 a patterns community outsider. *Inside* the patterns community I may be perceived as somewhat of a pattern geek; *outside* this community I talk with other developers and designers about their practices and techniques, design guidelines and heuristics, and successes and failures. Only occasionally do I mention patterns. By the way, I feel like an outsider to any community I am part of. This feeling isn't unique to the patterns community.

There are benefits to being an outsider. Because I don't feel singularly defined by any particular community, I find it comfortable to move between communities and learn new ideas. I'm not defined by pattern writing. Nor am I defined by Domain Driven Design, Agile development, software architecture, or Open Space communities. What I do care most about—and this transcends communities—is learning and sharing expertise and growing awareness and appreciation in others about how to sustainably design and build useful software systems.

Unfortunately, none of the patterns I have written have had much, if any, impact on the larger software development community. There hasn't been a broad diffusion of ideas or learning from the software patterns community to the larger software development community.

For various reasons my collaborators and I have not yet published these patterns in books. Perhaps if we had done so, that would give our patterns more legitimacy and make them more accessible.[1] But I suspect other factors also contributed to our patterns' obscurity—poor timing, a lack of approachability of written pattern forms to our targeted audience of practicing developers and designers, limited applicability, lack of promotion, or lack of any deep and lasting connections made between our patterns and other design practices, principles, patterns and schools of thought.

Additionally, there is the important matter of packaging for ready consumption. Some patterns we wrote were part of patterns collections. Instead of eventually being consolidated and housed in a book or focused website, they remain scattered in a series of papers published over several years in different PLoP proceedings which are obscurely stored in the ACM digital library. This makes them difficult for all but the most dedicated academic to find. Without more active curation, clustering our patterns into relevant sets that are discoverable by the people who could use them, our patterns will become lost.

My object design books [Wirf90, WM] have had a far greater impact.

This is in part due to the readability/approachability of these books, but mostly due to timing. Although I thought my second book on object design was an even more valuable contribution to design thinking than the first, it is the first book written in 1990 that was (and still is) widely recognized. When objects were "new," books on how to design object-oriented applications were eagerly sought after by software developers. By 2002, when my second object design book was published, object technology was well on its way to becoming mundane.

Timing is important. As is novelty. And luck.

But another force at play here that we should not ignore is the fact that inside the patterns community we are writing patterns at a meta level about software design. Most developers, however, are interested in the concrete and specific. They crave detailed design advice. Pattern descriptions—by intent—are abstract. There is a disconnect between the people I hope to reach and the way I am communicating my work through patterns.

Perhaps pattern literature is not the best way to convey software design knowledge.

Regardless, writing patterns has had a tremendous impact on me. My hope is that the following reflections will prompt you as software designers and developers and patterns authors to think more deeply about the relationship between written patterns and software design.

2. SOME LESSONS I'VE LEARNED THROUGH WRITING PATTERNS

Many software design and architecture patterns are only of interest to those working in narrow software niches. Toiling away writing about a variant of a particular pattern may contribute 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 these efforts are of limited value. Such knowledge, especially in a narrow field, adds value only if that knowledge can be readily shared among those working in that field. Not many designers today build adaptive object-model systems. Not many ever did. But there was a small core of contributors who wrote patterns for this architecture style in the early 2000s. I joined this group as a latecomer. I wanted to share my knowledge of using metadata in systems to design for flexible, interpretive behaviors. Plus, I like collaborating.

In hindsight, I caution potential authors working in a specialized software field: beware—the effort you take to document your work as patterns may not have any discernible impact on your field. Pattern writing may help you to understand and develop deeper design insights. 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

[1] Even though I've published these individual patterns papers on my website and in PLoP proceedings, any view of them as a coherent set is lacking.

exemplary solutions. But the dark truth is that publishing your knowledge only in pattern form limits the exposure of your work to those few who are either actively engaged in writing similar patterns or are academically motivated to seek out pattern literature (few academics are so inclined) in pursuit of making their own contributions. There is, however, value to you, the author of a pattern, as a note to your future self. Any broader impact is uncertain.

Pattern descriptions need refreshing.
Christopher Alexander and colleagues in the preface to *A Pattern Language* [AISJFA] observed that:

> *…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 therefore tentative, free to evolve under the impact of new experience and observation.*

Alexander and his colleagues expected patterns to evolve under new design contexts. What isn't so clear is how they expected new insights from other architects to be communicated or how their patterns could evolve. In *A Pattern Language,* Alexander only hints at creating project-specific pattern languages (emphasizing particular patterns and their importance) that would provide high level guidance for a specific architecture project*:*

> *All 253 patterns together form a language. They create a coherent picture of an entire region, with the power to generate such regions in a million forms, with infinite variety in all the details. It is also true that any small sequence of patterns from this language is itself a language for a smaller part of the environment and this small list of patterns is then capable of generating a million parks, paths, houses, workshops, or gardens.*

We software designers and architects also face the challenge of finding effective ways to update and refresh our patterns as well as to share project-specific design insights.

The need for pattern refreshing is especially true for most early software design "proto-patterns." When I first reviewed the *Design Patterns* book [Gamm], I hoped for a continuing stream of new design patterns from these authors. I strongly 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. The editors liked my suggestion and passed it along to the authors. But unfortunately, the original authors did not update their patterns for a variety of reasons, not the least being the untimely death of John Vlissides.

Admirably, others with no direct connections to these original authors or the patterns community are taking up this effort.

For one good example, consider John Thompson's web pages, which present the 23 patterns in *Design Patterns* refreshed with examples from the Java Spring Framework [Thom]. The descriptions are well motivated and provide an approachable introduction to these patterns that is especially relevant to Java programmers. The OBSERVER pattern is motivated by a more modern application (registering and receiving tweets from those you follow). And the example solution, more appropriately, uses Java interfaces instead of the original class-based solutions to define Observer and Subject implementable behaviors. To make the pattern even more relevant, a specific example illustrates how this pattern is applied in the Spring Framework. The author also thoroughly discusses the controversial SINGLETON pattern, demonstrating how to create a threadsafe version of a Singleton, and strongly expresses opinions on where it might be appropriate and why it should be sparingly used.

Another example is Brandon Rhodes' Python patterns guide website [Rhod]. In addition to showing examples of each of the 23 patterns found in *Design Patterns*, he includes additional common foundational Python patterns. He also discusses how one can come to a robust, comprehensive and flexible design solution by applying the design principles that underlie the patterns presented in *Design Patterns*. For example, in the discussion of the principle, favor composition over inheritance, different design solutions for logging are shown. These range from ADAPTER to BRIDGE and DECORATOR pattern implementations. In a section titled "Going beyond the Gang of Four patterns" Rhodes illustrates how Python's logging capabilities in its Standard Library are designed with even more flexibility—not only supporting multiple filters, but multiple output streams for log messages.

These two sites provide useful information for design-curious Java or Python programmers. They are at the level of detail that programmers can relate to, showing plenty of code along with thoughtful design commentary. None of their solutions are illustrated with the more abstract UML class or sequence diagrams.

Many broadly useful patterns 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 as an Open Textbook Library text.[2]

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

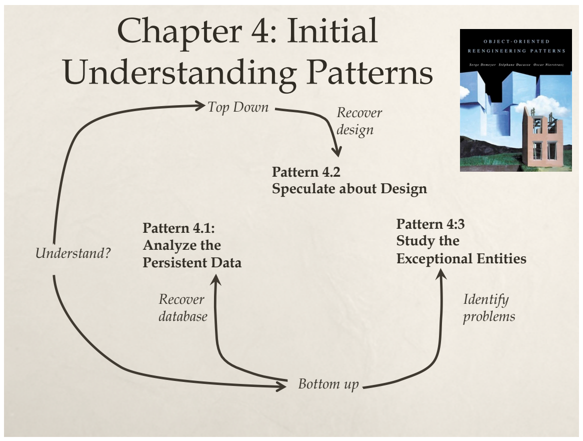
Chapter 4: Initial Understanding Patterns
Top Down
Recover design
Pattern 4.2 Speculate about Design
Understand?
Pattern 4.1: Analyze the Persistent Data
Pattern 4:3 Study the Exceptional Entities
Recover database
Identify problems
Bottom up

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

These patterns are still relevant in today's development context although they, too, could benefit from updating. 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 [Wirf17] 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 be rewritten to be generally applicable to both functional and object programming implementations, for example, MOVE BEHAVIOR CLOSE TO DATA, ELIMINATE NAVIGATION CODE, FACTOR OUT STATE, and FACTOR OUT STRATEGY. Perhaps if retitled *Software Reengineering Patterns,* with new chapters which described several functional programming language implementation patterns, this work could become even more relevant.

Another handful of small, useful, but unknown patterns comes from my own work. In a 2008 pattern paper [WY], we described three patterns for sustaining software architecture. Joe Yoder, my co-author, among other accomplishments, is known as a co-author of the BIG BALL OF MUD pattern [Foot]. That paper is notorious among software developers.[3] We wrote PAVING OVER THE WAGON TRAIL, a pattern for building a tool (most likely to generate code) that enables repetitive, error prone programming tasks to be eliminated. WIPING YOUR FEET AT THE DOOR is a pattern for cleaning up data by transforming it at the "edges" of a system in order to reduce internal complexity. A third pattern was about a well-intentioned effort to make programming simpler that instead

[2] https://open.umn.edu/opentextbooks/textbooks/object-oriented-reengineering-patterns

[3] Sadly, this paper is widely misunderstood. People commonly 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, if you actually read the Big Ball of Mud paper, you will find it contains a number of small, practical patterns for containing and managing complexity in a sprawling legacy system. These gems are 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.

reinforced poor programming practices and exacerbated the erosion of the architecture: PAVING OVER THE COWPATH. We took care to explain why this was not an anti-pattern, but instead an insidious form of well-intentioned tool tinkering. These patterns were intended to help prevent design decay and sustain complex, evolving architectures.

These patterns are still relevant. But as they never gained much visibility, they are at risk of fading into obscurity. Perhaps I humor myself by 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 limited with their naming. The BIG BALL OF MUD, after all, inspired us to give them names that implied muddy pathways, trails, and the need to clean data at system entryways. Those names detracted from their potential "stickiness." I suspect an even more important factor contributing to their insignificance was that we spent no time at all connecting our new patterns to other well-known related patterns such as Eric Evan's ANTICORRUPTION LAYER pattern [Evan] or the smaller patterns contained, obscurely, in the BIG BALL OF MUD. We certainly knew about these other patterns, but frankly we were more interested in writing new patterns that incorporated our shared knowledge than in hooking them to the broader body of existing patterns and promoting their recognition and use.[4]

Patterns, or small clusters of them—no matter how useful—cannot thrive in isolation.
Imagine having just read one of Alexander's patterns [AISJFA] such as the WINDOW PLACE in isolation. Without the awareness of the many other patterns found in *A Pattern Language* or a larger context of designing a dwelling, applying a single pattern won't have much impact. It is only when a designer sees how individual patterns are integrated within a larger body of design knowledge that she gains deeper insights and awareness of how patterns interact to create a holistic design. Not only do patterns need to be connected; there are naturally occurring larger "enclosing" patterns that also should be identified to provide a home and organizing structure for patterns with a smaller design scope.

To be relevant and stay relevant, software design patterns need to be richly connected to each other as well as to other useful design concepts, principles, heuristics and practices (regardless of their authors and whether they've been described in pattern form). These connections need to be strong ones. Connections need to be explained and made explicit. Connecting so deeply requires more effort; simply citing references to prior work won't suffice.

Let me explain the notion of making deep connections between concepts and patterns with an example from my own experience. Long before I wrote patterns, I conceived of the concept of object *role stereotypes* [Wirf92]. Role stereotypes are purposeful oversimplifications that can be used by designers to identify and reason about the work that objects do [WM, Wirf06].

Once I observed and named these six "patterns" of behavior in object-oriented applications—at that time, Smalltalk applications—I found I could spot them anywhere. In later works on Responsibility-Driven Design [WM, Wirf06], I demonstrated how well-known software design patterns embodied one or more role stereotypes. For example, a *coordinator* role is one where an object in a rote or mechanical way delegates work to other objects. The MEDIATOR pattern is an example of a *coordinator*. And the FAÇADE pattern is an example of an *interfacer* role where an object serves as an intermediary between different parts of the system. I also promoted the idea that by identifying patterns of interacting role stereotypes one could more clearly see a software design, whether emergent or existing. This way of seeing for me not only helped with design discussions and comparison of alternative approaches, but also with reverse engineering the design of existing object technology implementations. It was so important that I wrote another book about it and and updated a group of techniques for seeing and shaping an object design [WM]. Although I tried writing individual role stereotypes in pattern form, I quickly abandoned this effort. Role stereotypes were aids to perceiving the behaviors embodied in a design. That design may or may not incorporate any known design patterns. Regardless, I could use role stereotypes and "patterns" for arranging them, along with other design heuristics and principles to structure interacting object behaviors. Role stereotypes were more fundamental than software design patterns. In and of themselves they were not solutions to known design problems. So they weren't pattern worthy.

[4] This urge to create rather than integrate new knowledge shouldn't be surprising. None of original authors have updated the Big Ball of Mud paper or Domain Driven Design (DDD) patterns. Fortunately, others in the DDD community have written several books popularizing additional Domain Driven Design patterns and practices, distilling the essential DDD patterns, showing how they can be implemented in functional programming languages, and integrating them with other modeling practices.

Recently, Olaf Zimmerman and colleagues have written and published Microservice API patterns that were inspired by my notion of role stereotypes [ZPLZSa, ZPLZSb, ZSLZP]. Although our work has been cited in their patterns papers, it is a small footnote among several. Consequently, the reader of their patterns, especially as they are cast on their website, is not directly connected to idea of object role stereotypes which could be useful in designing the internals of services that implement these API patterns. Instead, designers will have to make those connections on their own—if they ever do. Finding relevant threads to pull on to make connections between newer and older design concepts is left as an exercise to only careful, studious readers or software historians.

Many software design and architecture patterns are overly specific and imply prescriptive technology solutions.
It is all too easy in 2020 to criticize the original *Design Patterns* descriptions as being outdated. But this isn't a fair assessment. 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 observed in multiple, pre-existing, successful software system implementations. *Design Patterns* was written when object-technology was gaining prominence and object design solutions were common (they still are, but are not as prominent).

Nonetheless, even the Spring Framework design patterns author offers a somewhat constrained view of these patterns' utility as evidenced by this introduction [Thom]:

> *The GoF [authors of Design Patterns] 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]

There is nothing technology 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, these patterns can be implemented in any programming language. Most developers today construe the original 23 *Design Patterns* as relevant only to object-oriented software solutions.

I know I can use an ADAPTER or BRIDGE or STRATEGY regardless of technology because I've done so. A more inexperienced designer might not so easily make this leap. However straightforward it is for me to transpose design patterns to different programming language and design contexts; I recognize that this ability comes from experience and seeing lots of solutions. Even though the original pattern descriptions might have been overly constrained or deceptively simplified, I can make that leap to another (similar enough) context.

It is noteworthy 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, more abstract representations of a software design solution—say a UML class or sequence diagram fragment—are becoming increasingly rare because such representations are not perceived as conveying useful additional information. Perhaps in this Stack Overflow era, most programmers are looking for solutions to immediate problems rather than looking to acquire generalized design knowledge.

There's a tension between presenting concrete solutions and generalizable abstractions.
However potentially useful software design patterns may be, unless they have been contextualized for today's technologies and software designers, it will be difficult for newcomers to understand and apply them. An accessible introduction to a specific software pattern and its significance ideally provides a context and a concrete example that can be easily grasped. Without any higher abstractions or lingua franca, when the technology changes—to stay approachable—that solution will need updating.

Paradoxically, it is the concreteness of some pattern solutions that deceive 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." So, when pattern authors present a realistic concrete solution, we readers may be lulled into thinking we know the extent of the pattern simply because we understand it.

---

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

To present the essence of a pattern—its abstraction if you will—requires a different type of exposition. That more abstract form needs a sparser description with a simple yet exemplary sketch of the problem. The solution shouldn't be spelled out in great detail. Or, if a solution is presented, it might be representative. A more abstract pattern should convey not only its significance but at the same time be sketchy enough that designers are forced (and expect) to fill in many details to fit it to their specific design situation. Equally important, that pattern needs to be located among other patterns and within a larger design context.

Both abstract and concrete descriptions are valuable—just to different audiences.

To date we, as a loosely formed patterns writing community, have not identified nor agreed upon conventions for tagging pattern descriptions with cautionary advice or tailoring them to specific audiences and contexts. For now, pattern authors are merely advised—at least in several writing workshops I have attended—to simply explain their choices and style of descriptions to their intended readers.

3. LOOKING 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 from requirements *into* design.

There are times when software designers don't see clearly 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 a philosophy of phenomenology, which firmly places us in a world of concepts, domains, phenomena and machines—in our case as software designers those machines are software mechanisms of our own creation—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. 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—around which you would eventually write requirements.

I recall, when first reading 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. The problem frames Jackson described didn't appear immediately relevant to the problems I frequently encountered in IT and software engineering. Jackson's frames seemed most appropriate for characterizing requirements for physical control systems. However, I found that with a little bit of mental effort that I could "stretch" his conceptual framework to fit problems I was designing for.

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

However, once I conceptually understood problem frames and how they could map onto my problems, I caught glimpses of them everywhere. Complex software systems interacting with other systems and databases tend to have multiple overlapping problem frames. 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. There was a set of software related questions relevant to each frame [WTN].

For example, here are some questions one might 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?
- 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)?

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" [WTN]. And while problem frames are firmly rooted in the problem space, to us they also suggested potential design 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 way, among many, to explore a problem space—a conceptual tool I use to focus on specific aspects of the problem as I untangle complex system requirements.

At the 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, as well as 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? Perhaps. Framing gave me yet another way of seeing problems. Problems are frequently composed of multiple overlapping problem frames. So the act of framing a problem doesn't straightforwardly lead to any particular design solution. The path from framing a problem to choosing an appropriate software architecture or set of design patterns is roundabout at best.

However, once you "see" a problem frame, applying the lens offered by that particular frame leads you to ask focused questions about a portion of software system's requirements. The answers to these focused questions don't directly lead to specific design patterns and approaches; but they do raise your awareness of what might be harder problems to solve.

We remarked in our patterns 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[6] of structure to resolve solution space forces" [WTN].

Did problem framing help me and others become better analysts? I'm unsure.

I found teaching others about problem frames to be an abject failure. Both the formal diagram notation as well as the concept of 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. My students didn't get the point.

They did, however, find it useful to have sets of related questions they could ask to gain further insight into their system's requirements. That those questions were related to the concerns relevant to specific problem frames was a distraction. 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 one might ask to get at those details) were practical skills that they could absorb and appreciate. The conceptual

[6] 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*.

backdrop of problem framing didn't need to be their focus. Rather, it was the hidden guiding hand behind those questions.

Framing is yet one more way to gain some perspective on a part of the problem at hand. It's useful to me as one of many tools, regardless of whether I can connect those insights with any patterned design solution. Once I know what the problem really is, then I can make use of my knowledge of both software design patterns and heuristics to work out a plausible solution.

4. SOME CONCLUSIONS ABOUT THE CURRENT STATE OF SOFTWARE DESIGN PATTERNS

As designers and architects, we aspire to readily connect known problems to plausible solutions. But unless I am designing something similar to what I have designed before, I must put in a lot of work before I gain any design traction let alone create a sustainable architecture. I can only proceed to design with confidence after a fair amount of experimentation and thinking. While I recognize that design patterns offer only general solutions—and that those solutions, depending on the author's particular writing style, are at varying levels of granularity and abstraction—I still find that I am frustrated when I apply new-to-me patterns. I have to fill in so many details and make many myriad smaller design decisions. Sure. I expect that. But it is precisely these extra bits of wisdom that I seek out when I enter new design territory—those particular design heuristics and tips and insights that come from lived experience. Pattern descriptions rarely convey those things. While I greatly value software design patterns—as they embody useful design knowledge—I have little confidence in their immediate, straight-out-of-the-box utility.

Even more disconcerting, newly published software design patterns are not often oriented to the existing body of software design patterns. And many useful software patterns remain relatively unknown. There is little coherence to existing software design knowledge, let alone the large body of software design patterns that are scattered about.

If you don't attend patterns conferences or regularly scour the patterns conference proceedings and published books or websites, you may not even find the good stuff let alone understand how it connects to or extends prior work. Consequently, you are reduced to making Internet searches and poring over Stack Overflow postings to find relevant design advice, one bite-sized morsel at a time, then piecing together the information on your own. While personally worthwhile, this doesn't serve the needs of the larger design community.

For example, as evidenced by a recent tweet stream [Garr20a], people are currently discussing implications of meshing the concept of Alistair Cockburn's HEXAGONAL ARCHITECTURE patterns [Co05] with the creation of Domain Driven Design's BOUNDED CONTEXTS and whether there should be new variants to the notion of PORTS and ADAPTERS. The threaded discussion started with a tweet announcing a new GitHub post on the topic [Garr20b]. It was followed by comments from another tweeter who also wrote a blog post [Pier] about his understanding of these patterns. Intermingled in the tweet thread were comments by the original pattern author. Eventually I discovered that the original tweeter also had a GitHub repository that contained several postings on the topic [Garr20c]. While I could piece together this conversation with the extra information, this simply reinforced my opinion that Twitter is horrible medium for capturing and preserving meaningful discussions and that pattern discussions are happening all over the Internet.

5. A CALL TO ACTION

If we as a patterns community want to promote software design pattern literacy, relevancy, and long-term impact, things need to change. Instead of staying the course, pumping out ever more patterns, I suggest we perform some bold experiments and learn from them how best to proceed with longer-term efforts.

Instead of only holding patterns conferences whose primary focus is encouraging new authors and students to write yet more patterns, we should devote a significant fraction of our attention toward discussing, organizing, updating, connecting and re-connecting, and rescuing the body of existing software patterns.

It may be that the existing knowledge is too sprawling to organize in any coherent way. And yet, in spite of the lack of any organized or concerted effort, the collective knowledge about patterns is growing. Perhaps our goal as a software design community shouldn't be to make everything neat and tidy. Design, after all is a messy process. Why should the evolution and integration of software design patterns, heuristics and practical design knowledge be different?

Acknowledging this messy state of affairs, we could sponsor or create an ecosystem that fosters patterns' continued use and evolution. In the early days of patterns, Ward Cunningham fostered open, wide-ranging discussions on design and patterns on his Portland Pattern Repository Wiki. Perhaps as a community we should gather together to recreate a curated space for discussing design and design patterns. I envision a lively forum where designers could share their insights into the many detailed decisions they make as they implement systems, and apply, extend, refresh, and newly conceive design patterns.

Additionally, we could hold software pattern mining, design heuristic hunting, and pattern refining events.

We in the patterns community can contribute our unique perspectives and a sense of our history to help weave together currently disjoint software patterns along with useful design advice and other design heuristics (whether written in pattern form or not).

We might even be so bold as to consolidate and present a "core" of what we believe to be the more enduring and impactful software design patterns.

Equally important, we should start publishing patterns workshopped at PLoP conferences differently. We could designate commentators to write an accompanying précis as well as an analysis that connects newly written and published patterns to the pre-existing body of knowledge. We might even speculate on new patterns' significance, scope, and utility. Or contribute additional examples of their use or exemplary solutions.

Chris Kohls and I [WK] recently pointed out yet one more avenue of exploration:

Rather than drowning pattern readers in even more text, verbal descriptions, and caveats, we propose that a better way to establish richer, more productive views of patterns would be to present curated views depicting multiple instances of particularly useful patterns in situ. Besides showing a good canonical implementation that applies a pattern, most patterns could benefit from 'how to not do it' code examples.

So, is it time for us to stop writing patterns? No, not entirely.

However, I think now is an opportune time for *me* to step away from cranking out ever more software related patterns to give myself the space and time to write, reflect, and work on the preservation, restoration, and promotion of important software design patterns as well as identifying gaps that need filling. This is no small effort. I cannot do this on my own.

Will you join me in these endeavors? We won't know of their impact unless we try.

6. ACKNOWLEDGEMENTS

I'd like to thank my shepherds, Stefan Hotel and Michael Keeling, for reading early drafts of this essay and prompting me to try harder to come to some conclusions and calls to action. I'd also like to thank my virtual writers' workshop colleagues for giving me pointed, if sometimes conflicting, advice on how make this essay more cohesive and compelling. In particular, I want to thank Lise Hvatum as well as Paulo Merson who provided me thoughtful comments even after the writers' workshop. And finally, thanks to my supportive and consistently thoughtful critics, Jordan Wirfs-Brock and 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.
[AHLSWY] Acherkan, E., Hen-Tov, A., Lorenz, D., Schachter, L., Wirfs-Brock, R., Yoder, J. (2011). "Dynamic Hook Points." Proceedings of the Asian Conference on Patterns of Programming Languages (AsianPLoP '11).
[Arn] Arnheim, R. *Visual Thinking*, (2004). University of California Press; Second Edition, Thirty-Fifth Anniversary Printing.
[Co05] Cockburn, A. " Hexagonal Architecture" Retrieved from: https://alistair.cockburn.us/hexagonal-architecture/
[Cunn] Cunningham, W. Portland Pattern Repository. Retrieved from: http://wiki.c2.com/?WelcomeVisitors
[Dem] Demeyer, S., Ducasse, S., Nierstrasz, O. (2003) *Object-oriented Reengineering Patterns*, Morgan Kaufman and [Website] open access textbook: https://open.umn.edu/opentextbooks/textbooks/object-oriented-reengineering-patterns.
[Evan] Evans, E. (2003). *Domain-Driven Design: Tackling Complexity in the Heart of Software,* Addison-Wesley.
[Foot] Foote, B., Yoder, J. (1997). "Big Ball of Mud." Proceedings of the Fourth Conference on Patterns Languages of Programs (PLoP '97/EuroPLoP '97). 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. (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley.
[Garr20a] Garrido de Paz, J. (@JuanMGP). 'Just published an article about the driven-side of #HexagonalArchitecture (pattern by @TotherAlistairrelated to "DDD Anti Corruption Layer". In the links section of the article I also mention some resources by @tpierrain about this topic.' 27 November, 2020, 12:25 PM. Tweet.
[Garr20b] Garrido de Paz, J. (2020) 'Hexagonal Architecture: The "Right Boundary"' https://jmgarridopaz.github.io/content/therightboundary.html
[Garr20c] Garrido de Paz, J. "Articles." https://jmgarridopaz.github.io/content/articles.html
[HNSWY10] Hen-Tov, A., Nikolaev, L., Schachter, L., Wirfs-Brock, R., and Yoder, J. (2010). "Adaptive Object-Model Evolution Patterns." Proceedings of the 8th Latin America Conference on Pattern Languages of Programs (SugarLoaf PLoP '10).

[HLNSWY10] Hen-Tov, Lorenz, D., A., Nikolaev, L., Schachter, L., Wirfs-Brock, R., and Yoder, J. (2010). "Dynamic Model Evolution." Proceedings of the 17th Pattern Language of Programing Conference (PLoP '10).
[HW15] Hvatum, L. and Wirfs-Brock, R. (2015). "Patterns to Build the Magic Backlog." Proceedings of the 20th European Conference on Pattern Languages of Programming (EuroPLoP '15).
[HW17] Hvatum, L. and Wirfs-Brock, R. (2017). "Pattern Stories and Sequences for the Backlog: Expanding the Magic Backlog Patterns". Proceedings of the 24th Conference on Pattern Languages of Programming (PLoP '17).
[HW18] Hvatum, L. and Wirfs-Brock, R. (2018). "Program Backlog Patterns: Applying the Magic Backlog Patterns". Proceedings of the 23rd European Conference on Pattern Languages of Programming (EuroPLoP '18).
[Jack] Jackson, M. (2001). *Problem Frames: Analyzing and structuring software development problems*, Addison-Wesley.
[Pier] Pierrain, T. (2020, November 29). Hexagonal or not Hexagonal? *use case driven.* https://tpierrain.blogspot.com/2020/11/hexagonal-or-not-hexagonal.html
[Rhod] Rhodes, B. Python Design Patterns. (April 2020). Retrieved from: https://python-patterns.guide.
[Thom] Thompson, J. Gang of Four Design Patterns. (April 2020). Retrieved from: https://springframework.guru/gang-of-four-design-patterns/
[Wirf90] Wirfs-Brock, R., Wilkerson, B., Wiener, L. (1990). *Designing Object-Oriented Software.* Prentice Hall.
[Wirf92] Wirfs-Brock, R. (1992) "Characterizing Your Objects." The Smalltalk Report, Vol. 2, Number 5.
[Wirf06} Wirfs-Brock, R. (2006). "Characterizing Classes." IEEE Software Design Column, Vol. 23, Number 2.
[WH16] Wirfs-Brock, R., Hvatum, L. (2016). "More Patterns for the Magic Backlog." Proceedings of the 23rd Conference on Pattern Languages of Programming (PLoP '16).
[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).
[WH18] Wirfs-Brock, R., Hvatum, L. (2018). "Even more Patterns for the Magic Backlog." Proceedings of the 25th Conference on Pattern Languages of Programming (PLoP '18).
[WH19] Wirfs-Brock, R., Hvatum, L. (2019). "Who Will Read My Patterns? On Designing a Patterns Book for Targeted Readers." Proceedings of the 26th Conference on Pattern Languages of Programming (PLoP '19).
[WK] Wirfs-Brock, R and Kohls, C. "Elephants, Patterns, and Heuristics." (2019). Proceedings of the 26th Conference on Pattern Languages of Programming (PLoP '19).
[WM] Wirfs-Brock, R., McKean, A. (2002). *Object-Oriented Design: Roles, Responsibilities, and Collaborations*. Addison-Wesley.
[WTN] Wirfs-Brock, R., Taylor, P., Noble, J. (2006). "Problem Frame Patterns: An Exploration of Patterns in the Problem Space." Proceedings of the 13th Pattern Language of Programs Conference (PLoP '06).
[WY] Wirfs-Brock, R., Yoder, J. (2012). "Patterns for Sustainable Architectures." Proceedings of the 19th Pattern Languages of Programs Conference (PLoP '12).
[WYG] Wirfs-Brock, R., Yoder, J., Guerra, E. (2015). "Patterns to Develop and Evolve Architecture During an Agile Software Project." Proceedings of the 22nd Pattern Languages of Programs Conference (PLoP '15).
[WYW07] Welicki,L, Yoder, J., Wirfs-Brock, R. (2007). "Rendering Patterns for Adaptive Object Models." Proceedings of the 14th Pattern Language of Programs Conference (PLoP '07).
[WYW08] Welicki,L., Yoder, J., Wirfs-Brock, R. (2008). "The Dynamic Factory Pattern" Proceedings of the 16th Pattern Language of Programs Conference (PLoP '08).
[WYW09] Welicki,L., Yoder, J., Wirfs-Brock, R. (2009). "Adaptive Object-Model Builder." Proceedings of the 16th Pattern Language of Programs Conference (PLoP '09).
[YWA2014] Yoder J., Wirfs-Brock R., Aguilar A. (2014). "QA to AQ: Patterns about transitioning from Quality Assurance to Agile Quality." Proceedings of the 3rd Asian Conference on Patterns of Programming Languages (AsianPLoP '14).
[YW2014a] Yoder J., Wirfs-Brock R. (2014). "QA to AQ Part Two: Shifting from Quality Assurance to Agile Quality." (2014) Proceedings of the 21st Conference on Patterns of Programming Language (PLoP '14).
[YWW2014b] Yoder J., Wirfs-Brock R., Washizaki H. (2014). "QA to AQ Part Three: Shifting from Quality Assurance to Agile Quality: Tearing Down the Walls." Proceedings of the 10th Latin American Conference on Patterns of Programming Language (SugarLoafPLoP '14).
[YWW2015] Yoder J., Wirfs-Brock R., Washizaki H. (2015). "QA to AQ Part Four: Shifting from Quality Assurance to Agile Quality: Prioritizing Qualities and Making them Visible." Proceedings of the 22nd Conference on Patterns of Programming Language (PLoP '15).
[YWW2016a] Yoder J., Wirfs-Brock R., Washizaki H. (2016). "QA to AQ Part Five: Being Agile at Quality: Growing Quality Awareness and Expertise." Proceedings of the 5th Asian Conference on Patterns of Programming Language (AsianPLoP '16).
[YWW2016b] Yoder J., Wirfs-Brock R., Washizaki H. (2016). "QA to AQ Part Six: Being Agile at Quality: Enabling and Infusing Quality," Proceedings of the 24th Conference on Programming Patterns of Programming Language (PLoP 2016).
[ZPLZSa] Zimmerman, O., Lübke, D., Zdun, U., Pautasso, C., Stocker, M. (2020). Interface Responsibility Patterns: Processing Resources and Operation Responsibilities. Proceedings of the European Conference on Pattern Languages of Programs 2020 (EuroPLoP '20).
[ZPLZSb] Zimmerman, O., Lübke, D., Zdun, U., Pautasso, C., Stocker, M. (2020). Data-Oriented Interface Responsibility Patterns: Types of Information Holder Resources. Proceedings of the European Conference on Pattern Languages of Programs 2020 (EuroPLoP '20).
[ZSLZPa] Zimmerman, O., Stocker, M., Lübke, D., Zdun, U., Pautasso, C. Microservice API Patterns. (December, 2020) Retrieved from: https://microservice-api-patterns.org/.