Elephants, Patterns, and Heuristics

REBECCA WIRFS-BROCK, Wirfs-Brock Associates
CHRISTIAN KOHLS, TH Köln

Capturing the wholeness of design solutions in order to effectively communicate them to others can be challenging. We posit that patterns are observable phenomena of design solutions. To represent these phenomena, a pattern author needs to generalize and omit information. Experienced designers are able to unfold the essence of the pattern and generate design solutions based on the information that they do find in a pattern description. Not surprisingly, their personal design heuristics play a central role in this. As they create a design solution, they also liberally apply their pre-existing design know-how and heuristics. But novice designers may have more difficulty. As this folding and unfolding of information and knowledge seems to be quite an abstract concept, we have chosen to make our point by discussing elephants. Like patterns, elephants are an observable phenomenon, a pattern in nature. Many different descriptions, representations, and accounts of elephants exist. Many people claim to know what elephants are. Yet they actually have little or limited knowledge of them. This analogy helps to understand how at the same time we both know and do not know what a thing is.

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. and Kohls, C. Elephants, Patterns, and Heuristics 26th Conference on Pattern Languages of Programming (PLoP), PLoP 2019, Oct 7-10 2019, 15 pages.

1. INTRODUCTION

Patterns are recurrent phenomena that can be found in all design domains. We can directly observe and experience these phenomena both as designers and users of those designs. However, it can be deceptively difficult to explain, generalize, and understand the salient aspects of these phenomena. The literature genre of pattern descriptions presents some form of a solution and explains other relevant information. To understand why a particular solution to a particular problem is reasonable, an explanation of the context where the pattern has been observed is described. Additionally, some understanding of the most critical factors that might drive a designer both towards and away from a particular pattern as well as potential consequences adds valuable perspective. This holistic analysis commonly found in pattern descriptions is based on Christopher Alexander's design theory as outlined in *A Pattern Language* [AISJFA] and *The Timeless Way of Building* [Alex].

Surprisingly, little discussion has taken place about the challenges pattern writers and researchers face when they try to preserve the wholeness of observed solution phenomena in written pattern descriptions. Since every pattern is a generalization, by necessity some details must be skipped. Pattern writers have to present what they consider the most significant information and insights in a way that is helpful to others. Other designers need to be able to unfold their unique solutions based on both these pattern descriptions and an understanding of their own design context and constraints. Each design situation is different. To unfold a specific solution from a general description requires tacit knowledge, experience, and the ability to make a series of tactical design decisions to achieve a satisfactory implementation.

The level of detail provided by any pattern description often depends on the target audience it was written for. Experts have a huge set of personal design experience and a wealth of heuristics they have internalized to draw upon. Patterns are just a small part of a much larger body of an expert's design know-how. Consequently, they are capable of unfolding new solutions from very general descriptions. Novices, on the other hand, typically need more guidance. Thus, pattern descriptions written for them tend to be very specific – up to an extent where they describe example solutions rather than more loosely constrained, generative patterns.

The information we have at hand or in our heads as designers in addition to the information provided in pattern descriptions are therefore critical factors.

And this is where elephants enter the stage. Like patterns, elephants are phenomena in the world. We can observe them, we can think we know a good deal about them, and there are many different ways to describe and represent elephants. Yet there are many misunderstandings, misconceptions, and missing information about elephants!

If we understand the challenges in capturing, communicating and using knowledge about elephants, we can gain insights into how to better capture and share software design patterns and heuristics. Because elephants are a pattern in nature. And everything we can say about the science of elephants, we can say about research into software design patterns and heuristics. Thus, in this paper we want to talk a lot about elephants.

2. ELEPHANTS ARE UNDER STRESS

"Elephant populations in India and also in the whole of Asia are under severe stress. The captive ones are rendered jobless due to changes in the mode of transport and lifestyle of people. The ones in the wild are also no better off, as the forests are shrinking." —Mark Shand

Are software patterns, too, under stress?

Most likely.

While captive, written software patterns initially proved useful to software developers, over time early software patterns either have been rendered irrelevant due to changes in technology and in the ways we design and build software or have become lost and mostly forgotten. While people may on occasion find software design patterns in online sources or be exposed to the idea of software design patterns at school, they tend to find design advice in other places. Most developers don't expect to read books on patterns to learn how to design software. They expect, instead, to learn from their experiences and their peers, supplemented by any online advice they may find.

Patterns written in the wild (those software patterns which have not been extensively reviewed at patterns writing conferences) tend to describe design solutions based on the latest software technologies or development practices. Information about a particular design topic can be scattered or contradictory, making it difficult to distinguish the important bits from the inconsequential. Other advice of varying quality can be found in blog posts and Stack Overflow replies, online courses, videos, and conference talks.

For example, while much has been written about the design of microservice architectures, only a fraction of this advice has been written in pattern form. The microservice pattern language written by Chris Richardson [Rich] in book form is also supported by a website (see www.microservices.io) that includes community discussion forums.

Another notable exception to pattern distress is the *Domain-Driven Design* [Evan] patterns. The initial patterns written by Eric Evans have been embraced and extended by a small, vibrant domain-driven design community. The building block patterns in Evans' book, e.g. the tactical object-oriented domain-driven design patterns, have been de-emphasized as the community thought leaders have turned their attention to the design of functional programming solutions and event-sourced architectures. However emphasis is still placed on the original "strategic" patterns for identifying larger structures and distilling core and generic sub-domains. Several authors in the domain-driven design community have refreshed the original design patterns with heuristics for identifying domain events [Bran], applying domain-driven design constructs to functional programming language implementations [Wlas], and designing event-sourced architectures [Youn]. Their focus has not been on writing their advice in pattern form so much as it has been on effectively communicating their current understanding and design practices.

While these efforts are notable, most patterns may be no better off than our community's carefully shepherded, tame patterns. Many "tame" and "wild" patterns are not sustained in a way that allows for their continued use, growth, and evolution based on feedback from a community of users.

3. CHALLENGES TO SHARING KNOWLEDGE

Ask different people "What is an elephant?" and they will give you different descriptions – yet all these descriptions are based on the same animals we label as elephants. You can describe an elephant with a few words, a sentence, a paragraph, a full page, or even in a book. The German Wikipedia article has 50,000 characters; the English Wikipedia article has 100,000 characters. Yet both articles write about the same phenomenon. And there are books about elephants that number in the several hundreds of pages.

Patterns and elephants share a lot in common. In fact, elephants are natural instances of patterns in our world. Each specific elephant exemplifies the general pattern of ELEPHANT. However, even for such a common category there are a lot of challenges to conveying a sound picture of what elephants are and how to treat them.

If we want to share our expertise, we need to communicate explicit knowledge about a particular phenomenon in some way.

We all think we know what an elephant is. But can we describe in a few words the nature of an elephant? If we face such challenges for elephants, how can we believe that sharing knowledge for the dynamic field of software development is any easier?

Likewise, a software pattern can be described on a single page—sometimes even a single index card—or on several pages. And, in the case of software patterns, not only are pattern authors challenged to simply describe the observed phenomenon; they also attempt to describe how to create a reasonable representation of that phenomenon, e.g. a stylized or exemplary design for that software pattern.

It is uncommon, however, to find books that describe a single pattern. For some social patterns, such as pedagogical patterns, sometimes there are books that discuss one pattern in length. For example, the pattern of a learning portfolio or the patterns of brainstorming have been described in typical pattern style as well as in dedicated books [Koh14].

Even if you do go into lengthy detail, each description of elephants is incomplete. One reason for this is that in addition to explicit information there are many implicit structures that are difficult to understand without seeing, or sensing on multiple channels an actual elephant.

Most descriptions of elephants are supported by visual representations. An image implicitly shows details that verbal descriptions can only explain in a roundabout way. For example, the spatial relationship of nose, eyes, ivory tusks, are shown as well as their relative sizes.

If we see a photograph of an elephant that represents the species of elephants in general, do we see an example of an elephant or do we see the pattern of elephants? The answer is both. Obviously a photograph depicts a specific exemplar of an elephant, an exemplification of the general pattern. However, we also see the pattern itself. A pattern manifests itself in each of its instances. Each instance shows the pattern.

Sometimes it is hard to discriminate this pattern because things in the real world are often manifestations of many overlapping patterns. Consequently, it can be difficult to distinguish the essence of the pattern and untangle it from the surrounding context.

For example, what is an elephant's skin color?

Asian elephants tend to have more mottled coloring, with skin tone ranging from dark grey to brown, with patches of pink on the forehead, ears, base of the trunk and chest [WWF].

African elephants typically have grayish skin: "The natural color of the skin is greyish black in both the African and the Asian elephant. To the observer of the elephant, the apparent color of the skin is determined by the color of the area's soil. This is due to the elephant's habit of throwing mud over its back " [EIR].

So what color are elephants? It depends on their context, their locale, and the color of any overlapping mud on their backs.

In software implementations, we often see classes that participate in different patterns with different roles in each of these patterns [BHS]. Pattern descriptions, however, focus on describing one particular pattern. In these descriptions the essential structure of a single pattern is isolated and an abstract representation is chosen.

Instead of showing a specific exemplar of an elephant, we could show a line drawing of an elephant. Unique features of a specific exemplar, such as shades of skin color or variation in ear size, may be left out in such an abstract representation. If we can clearly perceive an elephant, we have preserved its essential structure. In this case, the abstract representation still manifests the pattern. Of course, the drawing is not an elephant itself, but photographs are also not the elephant.

Representations that are too abstract, however, do not depict the pattern anymore: if you draw a box to represent an elephant, then the pattern is no longer discernible. If you think to label that box, "Elephant," then you are introducing an even more abstract way to represent an elephant; one without an obvious correspondence to its physical manifestation.

An alternative to abstraction is to use a model. A model is a good and valid instance that does not obfuscate the core structure or the essence of a pattern [Good]. For example, to represent an elephant, a picture (or 3D model) should show an archetypical exemplar of an elephant.

Likewise, the class diagrams that document software patterns are models rather than abstractions, because the more universal structure of the pattern is manifested in the concrete class diagram of that pattern. For example, the class diagram of the OBSERVER pattern has the same structural quality as a class diagram of an actual implementation; the details change—such as method names, parameter types, number of methods etc.—but the core structure is preserved. The interactions between the pattern elements are also universal between all instances of a proper OBSERVER.

And if the context is software design, creating a box and labeling it "Elephant" in a class diagram should not be confused with an abstraction of a physical elephant. Instead it should rightfully be interpreted as a representation of a software design element, perhaps with some correspondence to a physical elephant, but perhaps not. That label, "Elephant" has to be interpreted in the context of the design it is part of.

4. OUR PERCEPTIONS OF PHEONOMENA

Rudolph Arnheim, in *Visual Thinking* [Arn], explains three different *attitudes* or stances we take towards perceiving objects. These perceptual attitudes apply equally well to seeing elephants and software patterns. We label these ways of perceiving "contextually muddled," "contextually isolated," and "contextually integrated."

A *contextually muddled perception* happens when an observer "perceives the contribution of the context as an attribute of the object itself. …[The observer] sees, more or less, what a photographic camera records, either because he stares restrictively and unintelligently at a particular target or because he makes a deliberate effort to ignore the context and to concentrate on the local effect." [Arn] When the context changes, the object is observed as changing its character as well.

For example, we might observe a photograph of an elephant in a grassy savannah and that same elephant photographed standing underneath some trees. In the second photograph, the elephant is partially obscured by the shadows underneath the trees. Is the elephant now a darker color? To a naïve viewer with a contextually muddled perspective, an elephant's color would be perceived to change with a change in the landscape. But since we are a bit more sophisticated, and haven't heard tales about elephants that change their color so quickly, we'd answer: probably not. But that's based on our understanding of mammals in general, and their inability to rapidly change color.

Unfortunately, the essential nature of a software pattern can easily become much more muddled with its implementation context. What is it exactly that makes an OBSERVER an OBSERVER? In an implementation of OBSERVER there are two clearly distinguished roles that interact (the Observer and the Subject). If we implement the OBSERVER pattern in a programming language that supports the definition of Interfaces, we are likely to define Interfaces that are implemented by Observer and Subject roles. On the other hand, if we look to the original description of OBSERVER in the classic *Design Patterns* book [Gamma], we might conclude that any Observer is a subclass of an Observer class and the Subject must be a subclass of a Subject class. We've muddled the implementation details (e.g. the use of classes) with the essential behavior of Observer.

In the second way of perceiving—a *contextually isolated* view—the influence of the context is purposefully "peeled off in order to observe the object in its pure, unimpaired state." [Arn] The resulting object is constant, except for whatever changes the object initiates itself. Arnheim calls this a scientific way of viewing that "seeks to establish the nature of any phenomenon in itself in order to distinguish it in each practical case from the conditions surrounding it."

Perhaps, to get a contextually isolated perception of an elephant's coloring, we should scrub it clean of all dirt and mud and photograph it under bright light. But still we wouldn't get an accurate perception.

Most photographs of elephants show the front of elephant. But what about the back? What about a view from the top or bottom? We could observe an elephant photographed from different angles, removing it from its context, and thus piece together a more complete, yet isolated, depiction of that elephant. Is that better? Perhaps. But it is still incomplete.

Software design pattern solutions are commonly presented as contextually isolated. We see a static view of the structure of the solution in a class diagram. And sometimes, if the interactions between pattern elements are of interest, we are shown a dynamic view of those objects of the pattern interacting in a stylized sequence

diagram. Even though a slightly richer context where the pattern might be applied may have previously been explained in text, the solution itself is contextually isolated. We don't observe the solution embedded in any rich or realistic software context.

Joshua Kerievsky, in *Refactoring to Patterns* [Ker], talks about refactoring moves that either are towards or away from exemplary design pattern solutions. Joshua describes specific refactoring moves that turn an exemplary solution into slightly different forms. To perceive software patterns in the wild we need to be able to accurately spot them. The closer a pattern is to its exemplary description the easier it is to spot. But that exemplary solution isn't likely to be a reasonable design solution. Pattern solutions are always adapted to the current design context. The difficulty in perceiving a pattern, then, arises when a pattern changes form based on an unfamiliar-to-us context—we're simply not conditioned to perceiving it that way.

A *contextually integrated perception*, according to Arnheim, does not attempt to eliminate the effect of the setting on the object. Instead it fully "appreciates and enjoys the infinite and often profound and puzzling changes the object undergoes as it moves from situation to situation." Arnheim further claims that, "the enlightenment one gains from such varying exposure goes beyond aesthetic." [Arn] Observing an object or an elephant or a software pattern in novel situations often reveals fresh information.

How do elephants walk? How do they behave in specific situations? Video footage can cover more of these questions, it can even record sound. Showing a phenomenon in action is critical to understanding how it behaves in the world.

Even so, any visual depiction, even showing an object over a period of time, is still incomplete. Some phenomena such as smell or heat are not represented. Such information could be supplied verbally. But visual images do not show many details about the elephant's environment and social context. What about its relation to other elephants? How does an elephant integrate into its herd? What is its relation to other animals?

Likewise, we can ask for software patterns what other qualities do they have and how they relate to other patterns. This is often explained briefly in dedicated description fields in certain pattern forms. But rarely do we see depictions of the actual interplay of a pattern with other design elements in realistic software environments.

5. THE LIMITS TO WHAT WE CAN KNOW

"One of the reasons why they [elephants] have to consume so much food daily is due to their bodies. They only process about 40% of what they eat as the rest of it never gets digested. The digestion process for the elephant is very different than that of other animals. It really isn't understood why their bodies don't digest more of what they consume." [EIF]

What we can know depends on the questions we ask, our observations, and our actual experiences with elephants.

The answer to each conceivable question we can ask about elephants is more information about elephants. The answer is contained already *in the formation* of elephants, one only has to ask the question and find ways to capture the answer. For example, we can observe how much elephants eat and how much waste they produce still without knowing little if anything about how they digest their food: "Depending on species, elephants eat anything up to 350 lbs of plant matter on a daily basis" [Bio]. If you want to know the weight of an elephant, you have to put them on a scale. Both observations if properly done will not alter the nature of elephants—the facts about elephants have not changed. However, once we capture the answers, we have more information about the elephants. Still there are things we do not know yet about elephants after much observation.

Likewise, each software pattern contains an abundance of information. The specific information we capture depends on the questions we ask [Baey]. The typical format for pattern descriptions requires some information to be explicated. This explicit knowledge is the easy stuff; that which can be readily articulated, codified, and communicated. This format directs us to different questions and answers. Context, problem, forces, solution and consequences each ask different questions.

Often, for software patterns, there are more detailed questions as well, such as what are known uses, what are some implementation details, which roles can be identified? The more questions we ask, the more we learn about the solution. However, each pattern, and each of its actual implementations, always has more information than captured.

We should also acknowledge that there might be situations where the answer to a specific question is unknown. For example, we may understand what elephants are but do we understand in which environments they thrive? Can we enumerate or even generalize the contexts they fit into? Very often we see elephants in specific contexts: in films, in a zoo, in a circus, on a safari, at a specific time of the year. Can we really learn from these snippets which environments elephants belong in and in which environments they thrive? Paradoxically, we probably know more about elephants in environments other than their natural ones.

Designers face the same challenge when they observe software patterns. If a designer has employed a solution in similar situations, the context can be easily described. However, that designer may not be aware of other situations where the pattern is likely to fail, or for that matter where other suitable alternatives exist and might be preferable.

Consider the FAÇADE pattern. In the solution outlined in *Design Patterns* [Gamma] a single class implements the façade role. The purpose of that façade class is to hide implementation details and to present a simplified interface. However a façade can be useful for other (non object-oriented) technology solutions. And even in object-oriented solutions, a single class as the sole entry point might be considered over constraining or a poorly factored design under certain circumstances. In these days of more modern object-oriented programming languages, we might just as easily define one or more Interfaces as façades rather than define façade classes.

Investigating the problem and forces of a pattern can be compared to researching what has caused the specific form for elephants. What is their role within the ecosystem, how do they balance nature? How would their ecosystem destabilize if they disappear (problem!) and how does the specific organism of elephants fit to the environment (forces!).

Elephants do have a specific role and purpose within in their ecosystem even if science lacks knowledge about or misunderstands it. We have fully "functioning" elephants without us understanding all the details. And yet, there is much more to research about how elephants evolved and how they interact with their environment.

Likewise, we may further investigate problems and forces of existing software patterns. The problem and forces sections ask why-questions. Why do we use a specific form for a solution? A force explains the cause for a specific design decision by giving a "because" answer to the "why" question [Koh12]. But sometimes we know that a solution is appropriate (and usually works well) without being able to know exactly why it does.

So should we never write patterns before we have investigated all of the forces? Or should we just continue to search for more forces, better explanations, and deeper understanding of problems?

We speculate that patterns authors might hold off writing more definitive pattern descriptions until they experience or observe enough instances. What is precisely "enough" is hard to pin down. Indeed, that may be the wrong question to ask.

We view software patterns as a particularly informative form for describing design heuristics [Wirf17]. At the same time we are acutely aware that "[any heuristic] provides a plausible aid or direction in the solution of a problem but is in the final analysis unjustified, incapable of justification, and potentially fallible." [Koen] Using a particular heuristic does not guarantee design success. That's why we need to have at hand competing heuristics to try (and try again) if the particular design heuristic we choose fails us. Furthermore, we argue that we don't have to know all the reasons why some design pattern or heuristic (typically) works in order to apply it. We just have to have enough confidence that it is a reasonable fit for the current situation.

As we have seen, there are many different ways to describe and represent elephants, each varying in method and detail. So too, are there many ways to describe software design heuristics. Patterns are just one form. Probably a form we should reserve for those better-understood and appreciated design heuristics, even if we can't explain everything about them.

Science has established many different forms of representing phenomena of nature. On a walk through the Natural History Museum in London, I (Chris) found a section that curated different scientific methods to capture information and knowledge about animals, including observation, recording, mapping and modelling. At one end of the knowledge spectrum, we can zoom in to find more and more details. On the other end, we can zoom out to generalize and leave out information to focus on the core of a form.

A pattern solution may consist of the core solution (the thing), implementation details (the process), and things to take care of (liabilities). When we describe the solution of a pattern we also ask, what is the general structure of the solution? We can also zoom in and discuss specific details such as how to generate or

implement the solution, what variations exist, and where you as a designer have to be careful. We can also explore our understanding of the benefits, costs, drawbacks, trade-offs, and liabilities of the solution. Thus, we often ask explicitly about the consequences of a solution.

The more detailed the structure of our pattern descriptions are, the more information we provide. Sometimes as we attempt to write these descriptions, our knowledge gap becomes visible. We may not understand the whole of a context yet—even if we have successfully applied a solution multiple times.

Moreover, the general questions we ask about patterns can also be misleading. For example, on which level should we discuss problems? Many software patterns address the problem of implementing a specific design. Yes, it may be tricky to implement a well-behaved OBSERVER. So, it is important to present a reasonable solution. But there is a deeper understanding that a designer needs to have in order to truly "know" that pattern. What problem does an OBSERVER actually resolve? We are not only interested in how to solve the problem of implementing an OBSERVER. We are even more interested in what design problem an OBSERVER solves: to decouple reactive design elements from design elements whose state changes they react to. Hence, a good pattern solution should not only ask "How to do X?" in a problem statement. This "how to" is an implicit problem that must be addressed by each pattern anyways! This is because each pattern ideally should be generative and describe how to create a solution. But a pattern should also describe which actual problem is solved and answer the question why we need that specific pattern in the first place.

6. HOW CAN WE TRULY KNOW SOME THING?

"In captivity, elephants will eat pretty much anything so long as it is strictly vegetarian. They do seem to enjoy a wide variety of treats that they wouldn't find in their natural habitat – including pumpkins, and of course the famous peanuts! They're big fans of anything sweet and sugary, and love all kinds of fruit. Just like humans, elephants can have a 'favourite food' that varies depending on the individual – generally speaking it's some kind of sweet fruit." [Bio]

There are infinite numbers of questions we can ask about any elephant. Hence, there is an infinite amount of information we could gather about elephants.

To demonstrate the infinity of information about elephants, let us consider an example. A food designer wants to test two new elephant foods. The designer invents two new products. Now he wants to know whether elephants like A or B better. If we test this, we get the answer. The information is in the elephants already. But only by asking the question and observing their behavior do we get the answer and access to the information.

As there are an infinite number of potential new foods, there are an infinite number of questions we can ask about elephants. We can ask any kind of strange questions: Do elephants like to watch baseball? The answer is probably no, but you never know without testing!

As we seek out more information, we can find information that is not relevant to our situation. Whether A or B is more yummy for elephants may be of interest for a food designer or for the zoo management. However, most of us are more interested in general facts such as the amount of food or whether elephants ever eat meat.

Researchers and pattern authors try to hone in on the most "useful" set of information. Yet it is important to understand that the pattern of an elephant is so much richer than any representation can ever be. And that specific information may only be relevant in certain contexts.

We have seen that there is an infinite amount of information we can gather about elephants. Experts have access to a significant and relevant subset of this information. Many novices, however, think they have full information about a phenomenon when they repeatedly observe only superficial information.

Many people say they know what elephants are. But do they? Do they know their weight? How many offspring they typically have? Do they know how to react if they face an elephant in the wild?

We see the same with many software design patterns. Consider the MODEL-VIEW-CONTROLLER (MVC) pattern. Many developers learn about this pattern early in their education. There are more students who think they know what MVC is before they know what patterns are.

However, most developers reduce the MVC pattern to simply a concept that separates the model from the views and controllers, thus making the code structure more organized and the development of each design element more independent. There's nothing wrong with that. However, we see that even supposedly expert designers of frameworks implement MVC in many different ways that often violate core design principles. For example, the models, views and controllers are separated into different folders in the source code, but still

have many undesirable dependencies. And some (student) developers claim to follow the MVC architecture. But if you ask them how to add views or change existing views without changing the model they cannot provide proper answers.

Their superficial observation is that MVC separates the elements into different folders so developers can change the files independently. They miss the important design principle of loose coupling between objects and necessary abstractions. They also fail to recognize the OBSERVER pattern as a mechanism to notify the views about any changes in the model. Students think they know what the MVC pattern is without knowing and understanding the Observer mechanism which is used to achieve loose coupling between models and views, or more generally, between some design element that is observed and its observers.

This kind of superficial understanding of patterns is like saying: "I was in the zoo last week and watched the elephants for a day. Believe me, now I know all about elephants."

7. SOME DIFFERENCES BETWEEN EXPERTS AND NOVICES

"We habitually observe by the method of difference. Sometimes we see an elephant, and sometimes we do not. The result is that an elephant, when present, is noticed. Facility of observation depends on the fact that the object observed is important when present, and sometimes is absent." —Alfred North Whitehead, *Process and Reality* [Whit]

Seeing a pattern in action does not make us experts on that pattern. We need a deeper understanding. We need to see that pattern as it is applied in various situations, and to understand what the design becomes when other heuristics are used instead. We need to understand the design principles and values that led to that pattern.

We find that novices don't gain such understanding by reading the same material that experts consume. Why is this? Experts fill in the gaps in pattern descriptions with their own personal design heuristics and additional tacit knowledge they've acquired over time. Experts "see" more than is described. And they do so intuitively, without conscious effort.

Still the question remains: how might patterns best be introduced to novices?

The information provided in a pattern description needs to fit the knowledge and preferences of the target audience.

In order to understand a pattern, a novice has to contrast it with its surroundings before she can see what's important about that particular pattern. Additionally, she has to observe what's constant about that form over space and time and various use cases.

If design novices are also students, then one plausible pedagogical approach might be to introduce them to a particular design problem. Then, show them code that solves that problem using a specific pattern. After seeing that code and understanding what it does, then and only then explain the pattern to them.

To understand a particular pattern's significance, beginning designers also can benefit from seeing this pattern as it is applied in various different programming languages and technologies. And then, to gain a broader appreciation of that pattern's applicability, they could be given additional problems that they can solve by applying that same pattern. Repetition accompanied by meaningful, explicit variation.

Arnheim notes that a concept from which everything is subtracted but its invariants facilitates definition, classification, learning and use of that learning because "[t]he object looks the same, every time it is met." [Arn] Seeing a single concrete example is often not enough to comprehend a general abstraction. Designers need to observe slight design variances that can still be called some particular pattern so that they come to know both what this pattern means and what it means to not be that pattern. Sadly, stripped-down, the essential depiction of a solution for the typical software design pattern removes us from any concrete, realistic, tangible experience. The rigidity of such exemplars can blind novices to the idea that they should examine their particular context for insights and constraints. Designers should always expect to adapt the exemplary solution appropriately for *every* context. They need to see variations in implementation of a particular pattern. Beginners need to learn that there is no one "right way" to apply a particular pattern.

To become proficient at applying a pattern to solve a non-textbook or classroom problem, novices need to do more than a cut-and-paste reuse of a pattern. They also need to be exposed to other reasonable design solutions to the same problem.

This is something I (Rebecca) exposed my object design students to as a matter of course. I would show them several different solutions that the students themselves came up with. To do this, they solved a non-trivial problem. I hoped that they would choose to use the specific pattern they had just learned about to solve a problem (so we could observe their various implementations), but this wasn't always the case. Instead of insisting that they solve the problem a particular way, I allowed them to submit solutions as long as their solution worked reasonably well. We would then review and critique the various solutions. By seeing multiple design solutions that worked more or less, students came to appreciate the variety of reasonable design alternatives available to them, each with their strengths and weaknesses. To learn how to exercise design judgment, they also need to see and appreciate the nuances of different design solutions.

Arnheim argues that the kind of concept created by contextually integrated viewing (in this case, seeing different solutions for the same problem and then critically examining them) allows for productive thinking, that is, thinking "outside the box."

Experts appreciate knowing when to use a pattern (and advice on when not to use it) as well as how it compares with other design alternatives. There are always competing heuristics and multiple ways to solve any particular design problem. Experts also appreciate details: what complexities there are, what to expect during implementation, what variations there are and when choosing those variations might be preferable.

Many times, however, regardless of expertise, designers just want to solve the current pressing problem of the day. They don't want theory, principles, patterns, or lofty design heuristics. They don't have the patience to wade through explanatory information. Often they are seeking concrete, detailed, and very specific information or advice on what to do next. They don't need patterns so much as they need detailed relevant information that will help them take that next design step. Often this is at a different level (e.g. specific coding details) than most software patterns are written. Pattern descriptions, after all, are abstractions.

In this situation they seek an "exemplary" solution (ideally code) that they can read and quickly understand, in order to copy and modify to suit their purpose (e.g. taking code snippets from blog posts or stack overflow). Occasionally, especially more experienced designers may also want to know a bit about the why behind the recommended "what to do." But not until they've figured out that this particular solution seems reasonable.

But designers do take occasionally take a breath, and pause to think on what they did and how well it worked.[1] We suggest that recording one's design ideas and heuristics and experiences (not patterns) in a design journal is one powerful way to grow one's reflective design skills and expertise [Wirf18]. We need to reflect on the outcome of our design choices in order to learn from our experiences. Through the act of recording the design heuristics we applied (and under what situations they worked or didn't) we can become more intentional as designers, and more aware of our design choices and their consequences. We make choices based upon our specific context, our personal preferences, our experience, and our wealth of personal design heuristics that we have at hand (in addition to all we've read about or studied at school). Our tacit knowledge enables us to adapt patterns to our specific situations. And it will enable us to find creative solutions to design problems we've never seen before.

As Whitehead observes: "The true method of discovery is like the flight of an aeroplane. It starts from the ground of particular observations; it makes a flight in the thin air of imaginative generalization; and again it lands for the renewed observation rendered acute by rational interpretation." [Whit]

8. MISPERCEPTIONS, MISCONCEPTIONS, AND THE STORIES WE TELL OURSELVES

A little expertise can be dangerous. Thinking oneself an expert can be dangerous. If your assumptions about elephants are based on TV shows, zoos, and circus visits you likely will have constructed a wrong mental picture. If you face an elephant in the wild you may react all wrong. You may provoke the elephant in spite of its friendly nature.

Let us assume somebody wants to become an expert on elephants and studies them for a day in the zoo. She observes that elephants get their food at 9 a.m. in the morning from the zookeeper. So she claims that "Believe

[1] In my early years as a software engineer at Tektronix, I (Rebecca) was issued an engineering notebook, as were all electrical, mechanical, and software engineers. We were instructed to write down by hand (and to initial and date each page) our design ideas. This notebook's primary purpose was to document design ideas that were potentially patentable, but arguably, the discipline of writing down my design notes also helped me appreciate the value of clarifying and explaining to myself (and eventually others) my design ideas and approaches.

me, elephants start eating at 9 a.m. with food provided by from zookeepers. That's the nature of elephants." And if you raise some doubts, our elephant expert says: "Wait a few weeks and I will provide more evidence." She then observes elephants for the next 30 days at the zoo. And guess what: the elephants always get their food starting at 9 a.m. More precisely, the observations have shown some variations. Sometimes elephants get the food at 9:05 or even at 9:10, sometimes even at 8:58 a.m. So, our expert adjusts the statement and claims that elephants get their food between 8:58 and 9:10 a.m. She provides a lot of evidence based on 30 days of observation. Doing this over a full year, this data becomes statistically sound. What's missing?

The answer is: our expert has ignored the specific context. The claim that elephants start eating at 9:00 a.m. on every day is only valid for the specific zoo she visits. If she visits other zoos, she will learn about further variations. For example, at the Portland Zoo (which doesn't open to the public until 9:30 a.m.) elephants are fed at timed feeding stations. Zookeepers offer food to the elephants at specific times (after all, the typical elephant needs to eat nearly 200 kilograms of vegetables and grasses each day). When I (Rebecca) visited the Portland Zoo shortly after it had opened, I only observed elephants actually finishing up a timed meal, at 10:30. I do not know if they started eating at 9 a.m. and I do not know when they stopped eating (as I only observed them for an hour).

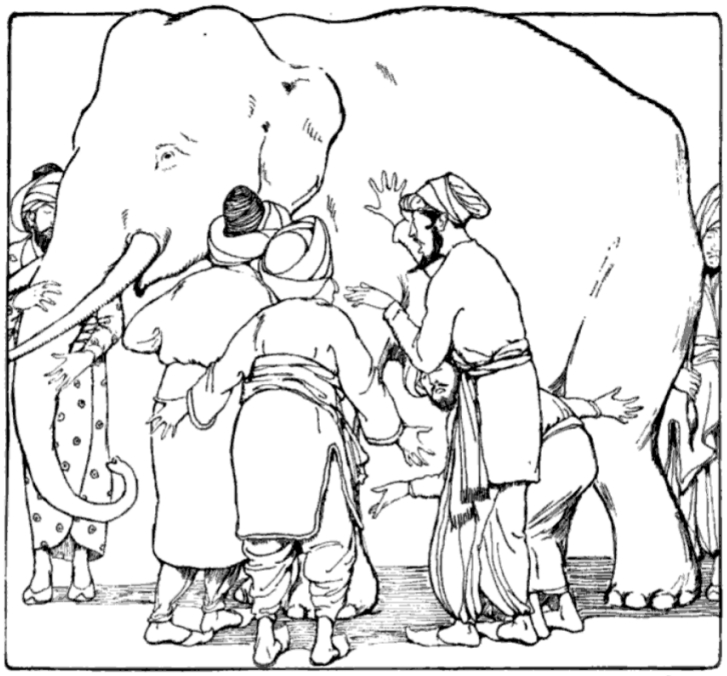
If our supposed elephant expert visits the same zoo 2 years later she may also see new data if the zoo management has shifted feeding times. And if our elephant expert would bother to observe elephants in the wild—in their most common context—she would make quite different statements about food habits.

Misinterpreting the scope of a context is common mistake. We observe a software design pattern within one specific environment: one company, one programming language, or one developer team, and assume that the characteristics of the pattern are true for many other contexts. However, without observing the pattern in these other contexts, we cannot make decisive statements about these other contexts! And most certainly, our perceptions may be obscured by any personal design heuristics and adaptations of that pattern for specific design contexts.

There is a difference between a single writer reporting her patterns and the outcomes of a group discussion. Likewise the range of domains and contexts in which a pattern has been observed is critical to its general applicability [KP]. For example, if a pattern has been observed multiple times in Java programs, does this necessarily imply that it will work for C++ code as well? Without having observed or tested it, one cannot really (empirically) tell.

The story of the blind men and the elephant is a very old parable that discusses the limits of perception and the meaning of context. It can be found in Buddhist, Hindu, and Jain texts (see Wikipedia). The parable goes like this (from Wikipedia) [Wiki]:

> A group of blind men heard that a strange animal, called an elephant, had been brought to the town, but none of them were aware of its shape and form. Out of curiosity, they said: "We must inspect and know it by touch, of which we are capable." So, they sought it out, and when they found it they groped about it. In the case of the first person, whose hand landed on the trunk, said, "This being is like a thick snake." For another one whose hand reached its ear, it seemed like a kind of fan. As for another person, whose hand was upon its leg, said, the elephant is a pillar like a tree-trunk. The blind man who placed his hand upon its side said the elephant, "is a wall." Another who felt its tail described it as a rope. The last felt its tusk, stating the elephant is that which is hard, smooth and like a spear.



By Illustrator unknown - From Martha Adelaide Holton & Charles Madison Curry, Holton-Curry readers, Rand McNally & Co. (Chicago), p. 108, Public Domain, https://commons.wikimedia.org/w/index.php?curid=4581243

What does this parable mean for pattern writers? We need to be aware that we may experience parts of a pattern but are still missing important aspects. We need to be careful to not call ourselves experts too soon. It also shows that we can report and describe the very same things in quite different ways when we focus on different parts of the whole.

We have argued that pattern descriptions try to capture forms that exist in the world. Whether a pattern description adequately captures a structure that can actually be found in the world is an outstanding question. Moreover, if the authors are not experienced in their domain they might capture the wrong patterns. But even if we have adequate patterns in our head, the explicate description will always be incomplete, misleading, or focus on the wrong things.

Therefore, we need ways to ensure that the pattern descriptions actually represent meaningful patterns of the world. A written pattern should be the result of thoughtful pattern mining, a process that extracts "nuggets of wisdom." We can say "wisdom" because the insights in those patterns are grounded in many reviews of actual designs. Extracting this knowledge is like mining for nuggets [Ris]; the core of the pattern is pointed out; the noise of actual instances is taken away. This mining process is a process of cognition—as is any theory building. A pattern author often reconsiders the artefacts and examples her pattern is based on. Writing down the pattern is also an active process whereby the writer tries to assemble the universal structure of the pattern. Pattern descriptions are proposals of specific views on the world, and of solutions to agreed-upon design problems in particular. However, their validity needs to be tested, as does any theory.

9. LET'S TALK ABOUT THE ELEPHANT IN THE ROOM

An "elephant in the room" is "an important or enormous topic, problem, or risk that is obvious or that everyone knows about but no one mentions or wants to discuss" (Cambridge academic content dictionary). There are various reasons an "elephant in the room" might not be addressed. The issue might be that the topic is: something that is uncomfortable to bring up, or something that is taboo, something we choose to deliberately ignore, or something that is so obvious to everyone that we don't mention it.

We believe that there are several elephants in our patterns community room. They exist for various reasons. *Nobody has been talking very much about these elephants in the room.* Or if they are, they haven't been speaking loudly. This essay has touched on some of these issues and elephants in the rooms. We want to

conclude on a positive note by summarizing them and then offering some avenues for patterns authors, educators, and software developers and designers to explore.

Something that is uncomfortable to bring up: To get the "right" views on patterns is problematic. Perhaps this shouldn't be our goal. Instead, we need to seek augmented ways to depict software patterns that allow for productive thinking and their creative application.

There are many ways of seeing the world and organizing its structure; and there is always doubt as to whether we have seen enough of the world to identify sufficiently stable patterns (let alone good examples of them). Patterns that have been identified in a pattern mining process are fallible in principle and can be falsified empirically. If a pattern consistently fails, even if it is well known, it needs to be rejected—perhaps the pattern description, or the pattern in its entirety. However, patterns, like all design heuristics, are fallible. Successful pattern applications, on the other hand, are corroboration of the adequateness of the insight provided in a pattern description.

Pattern descriptions, or any other account of design heuristics, are not simply about finding the "truth" about good design. They are also design tools to generate new good designs. Thus, they go beyond ordinary theories. The quality of writing of a pattern and the ways patterns are depicted matters just as much as does the adequateness of the identified pattern.

Many people invoke the metaphor of a story or a play [Ris] and point out that patterns are not just about facts but should tell a story [Appl]. The context sets up the stage. As in a play, the forces are creating a tension and the solution is resolving the conflict—a happy ending.

However, is this sequence always the appropriate order? There are many ways to tell a story, and some stories start with the end. If we were about to describe an elephant, would we naturally start with all his evolutionary history and reason about why this species fits into the very environment elephants live in? Or, would we start with a picture first and then go into details? Most accounts about elephants use the latter approach. First show the object, and then explain the phenomenon.

Stories are such a powerful tool because they are capable of transporting the wholeness of a solution. We experience wholeness if we follow a story in a novel. The plot unfolds chapter-by-chapter, paragraph-by-paragraph, sentence-by-sentence, and word-by-word. The parts make the story and the story gives meaning to each of the parts. A simple sentence such as "The door was locked" has its own meaning; however, in the context of a larger story its meaning can shift. A locked door has a deeper meaning in a crime story where a victim tries to escape. The same sentence can have a different meaning in a love story: "She wanted to tell him her feelings and caught up with the train at the local station. The door was locked." The context not only changes the meaning of the sentence; the single sentence that reveals an important fact or event can also change the meaning of the whole story. The story directs the development of the events, scenes and characters; at the same time the story is made up exactly out of these interrelated parts.

If we consider the literature genre of patterns as storytelling, then we should allow and encourage different forms of telling this story: A short story; a whole book; a series of stories; or even telling the story with motion pictures or cartoons.

Something that is taboo: Software patterns today are often irrelevant and invisible. The body of software patterns over time has become disorganized, stale, outdated, and unknown to many software developers. While new patterns are being written, they are mostly ignored unless they are about a popular trending technology (e.g. microservices, event-sourced architectures). When new patterns are published they aren't located within the overall pre-existing software pattern landscape. Consequently, they are disconnected from prior work. There is no overall coherence to the large body of software patterns. Furthermore, there's a wealth of useful, specific, concrete design advice being written and communicated to broad audiences that are not written as patterns.

If we want to encourage pattern literacy, relevancy, and long-term impact, something needs to change.

In addition to finding better ways to organize patterns and presenting relevant depictions of them, we need to find better ways to explain how to adjust patterns into specific contexts, how to sort through them, and add or find the information a designer needs when she is able and willing to absorb it.

Furthermore, the word pattern may be the wrong word for what we write. Most people outside of software think of patterns as being something different than what we create when we write our software patterns.

Merriam Webster lists several meanings for pattern. The first definition is "a form or model proposed for imitation, an exemplar." The second definition is "something designed or used as a model for making things."

While our software patterns often provide simple solutions, should they be exemplars or models for makers? Or should they be something less, e.g. design gists?

Something deliberately ignored: Different audiences for software design patterns need different ways to absorb, comprehend, and understand how to apply them.

The way experts absorb information differs from novices. Historically, software patterns were written for practitioners (and experienced ones at that) and not novice developers. Ironically, it is the original 23 patterns in *Design Patterns* [Gamma], written by experts for experienced developers that were and still are taught to novices.

Researchers find that experts, while they may not agree, are logically self-consistent in their individual opinions [Shan]. Patterns written by single individuals or a tight-knit group of collaborators are cohesive. Patterns from different sources often are not. Experts can absorb differences of style and substance with some effort and even reconcile conflicting patterns' advice; this is a much harder task for those new to patterns or to software design.

If we want many more software patterns to have a lasting impact, there needs to be some way that a broader, more useful number of software patterns are coherently organized and easily found.

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.

Additionally, designers should be encouraged to create and share notes on their personal experiences with patterns. What if there were easy ways for designers to annotate and enrich published patterns with their own design heuristics? How to best accomplish this (and which patterns warrant such curation) is a topic for future research. Focusing solely on writing new patterns misses an opportunity for the patterns community to hook up patterns with others' real world experiences.

An obvious problem: Currently, we mostly write patterns for ourselves or for people whom we think are like us. This limits our patterns' reach and impact. Pattern forms need to be refreshed.

Arnheim argues that the kind of concept created by contextually viewing objects is better suited for reasoning about those objects in different situations and under different conditions. Our current written forms for software patterns fall short in this regard—pattern depictions typically describe just enough context and forces, before providing a stylized, exemplary solution. For the most part, pattern solutions present a contextually isolated view. While this facilitates definition, classification, and learning, it does not build in the mind of the reader a deep understanding of that pattern in a realistic setting. Simply reading patterns and learning pattern names doesn't ensure that the reader can apply them appropriately (if at all). Moreover, learning about patterns may be counterproductive to learning how to exercise design judgment

We patterns writers, too, could benefit from further stretching our imaginations to envision the boundaries and true shape of our patterns. By mentally exercising the limits of our patterns we might gain new insights. Again, from Whitehead: "The reason for the success of this method of imaginative rationalization is that, when the method of difference fails, factors which are constantly present may yet be observed under the influence of imaginative thought. Such thought supplies the differences, which the direct observation lacks. It can even play with inconsistency and can thus throw light on the consistent, and persistent, elements in experience by comparison with what in imagination is inconsistent with them. This negative judgment is the peak of mentality." [Whit]

As patterns authors, we intentionally create depictions of what we have found along a design journey that we've already taken that we hope others can follow. But we shouldn't be content to only write in pattern form. Patterns convey critical information so that others on similar journeys can learn about our design thinking.

We have an opportunity to offer our fellow designers much more.

What if we were to tell more of our personal story as designers and pattern miners? We might describe what systems we've designed or seen, and under what conditions they were designed. Or share how we discovered our software patterns and enumerate other potential intriguing design ideas that we spotted or were aware of but didn't include (and why).

We might share where we'd like to investigate further—other design contexts where we are curious, or not—places where we are cautious or reluctant to recommend using our patterns. We might experiment with recording others' heuristics that fill in the gaps, conflict with, augment, and mesh with our written patterns. We

might share how confidant we were about our patterns' utility or our perception of their relative value and whether that perception has changed over time. Or we might be so bold as to rate our patterns with recommended design experience required to utilize them successfully. While all this stuff is "outside" our current pattern forms, we think it this, too, is important information for designers to know.

10. ACKNOWLEDGEMENTS

We'd like to thank our shepherd, Joseph Yoder, for reading early drafts of this essay and prompting us to try harder to weave heuristics and elephants into it. We'd also like to thank our writers' workshop colleagues for giving us pointed, if sometimes conflicting, advice about ways to make this essay more cohesive and compelling. In particular, we want to thank Lise Hvatum who continued to provide us thoughtful comments after the writers' workshop.

REFERENCES

[Alex] Alexander, C. 1979. *The Timeless Way of Building*. New York: Oxford University Press.
[AISJFA] Alexander, C., Ishikawa, S., Silverstein, M., Jacobson, M., Fiksdahl-King, I., & Angel, S. 1977. *A Pattern Language*. New York, USA: Oxford University Press.
[Appl] Appelton, B. 2000. Patterns and Software: Essential Concepts and Terminology. http://www.cmcrossroads.com/bradapp/docs/patterns-intro.html. (1.7.2009)
[Arn] Arnheim, R. 2004. *Visual Thinking,* University of California Press; Second Edition, Thirty-Fifth Anniversary Printing edition.
[Baey] Von Baeyer, H. C. *Information: The new language of science.* 2004. London: Phoenix.
[Bio] BioExpedition, Elephant Feeding. Retrieved from: https://www.bioexpedition.com/elephant-feeding/
[BHS] Buschmann, F., Henney, K., & Schmidt, D.C. *Pattern-oriented software architecture. Volume 5: On patterns and Pattern Languages*. 2007. West Sussex: John Wiley & Sons.
[Bran] Brandolini, A. 2019. *Introducing Eventstorming,* LeanPub.
[EIR] Elephant Information Repository. The Skin. Retrieved from http://elephant.elehost.com/About_Elephants/Anatomy/The_Skin/the_skin.html
[Gamma] Gamma, E., Helm, R., Johnson, R., Vlissides, J. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley.
[Good] Goodman, N.1976. *Language of art: An approach to a theory of symbols.* Indianapolis, Ind: Hackett Publishing Co.
[Ker] Kerievsky, J. *Refactoring to Patterns.* Addison-Wesley, 2004
[Koen] Koen, B.V. 2003. *Discussion of the method: Conducting the Engineer's approach to problem solving,* Oxford University Press.
[Koh12] Kohls, C. "The Path to Patterns - Introducing the path metaphor". 2012. EuroPLoP 2012. – 17th European Conference on Pattern Languages of Programs. New York: ACM.
[Koh14] *The theories of design patterns and their practical implications exemplified for e-learning patterns.* 2014 https://opus4.kobv.de/opus4-ku-eichstaett/files/158/kohls_patterns13032014.pdf
[KP] Kohls, C., & Panke, S. Is that true? "Thoughts on the epistemology of patterns." 2009. Proceedings of the 16th Conference on Pattern Languages of Programs. New York: ACM.
[Evan] Evans, E. 2003. *Domain-Driven Design: Tackling Complexity in the Heart of Software,* Addison-Wesley.
[Rich] Richardson, C. 2018. *Microservices Patterns,* Manning.
[Ris] Rising, L. 1998. *The Pattern Handbook*. Cambridge: Cambridge University Press
[Shan] Shanteau, J. 2001. "What does it mean when experts disagree?" in E. Salas & G. Klein (Eds.), *Linking expertise and naturalistic decision making* (p. 209–244). Lawrence Erlbaum Associates Publishers.
[Wirf17] Wirfs-Brock, R., "Are Software Patterns Simply a Handy Way to Package Design Heuristics?" 2017. PLoP 2017, Proceedings of the 23rd Conference on Pattern Languages of Programs.
[Wirf18] Wirfs-Brock, R. "Traces, Tracks, and Trails: An Exploration of How We Approach Software Design." 2018. PLoP 2018, Proceedings of the 24th Conference on Pattern languages of Programs.
[Whit] Whitehead, A. 1929. *Process and Reality. An Essay in Cosmology. Gifford Lectures Delivered in the University of Edinburgh During the Session 1927–1928.* Macmillan, New York, Cambridge University Press, Cambridge UK.
[Wiki] Wikipedia, Blind Men and an Elephant. Retrieved from https://en.wikipedia.org/wiki/Blind_men_and_an_elephant
[Wlas] Wlashin, Scott. 2018. *Domain Modeling Made Functional: Tackle Software Complexity with Domain-Driven Design and F#,* Pragmatic Bookshelf.
[WWF] World Wild Fund for Nature, Asian Elephants. Retrieved from https://wwf.panda.org/knowledge_hub/endangered_species/elephants/asian_elephants/
[Youn] Young, Greg. 2017. *Versioning in an Event-Sourced System*, LeanPub.

APPENDIX: KEY TAKEAWAYS

A design heuristic according to Billy Vaughn Koen is “anything that provides a plausible aid or direction in the solution of a problem but is in the final analysis unjustified, incapable of justification, and potentially fallible.”

Patterns are a particularly useful form of design heuristic.

The vast majority of software design heuristics have not been written in pattern form. These heuristics exist as tacit knowledge in the heads of designers gained from experience. Some have been written (mostly informally) in many other sources.

The patterns we observe in a software design are much richer than any documentation or visual depiction could ever convey.

A pattern description is just one view of the design phenomenon.

Software patterns are particularly useful as they are drawn from direct experience and include useful information for the discerning designer—most notably the context where the pattern has been found to be useful, an exemplary solution, as well as some tradeoffs and consequences of applying it.

The standard description format for patterns helps us to ask the right questions about a good design. However, there are other useful ways to describe the phenomenon.

Describing the forces and consequences helps us to understand how and why a pattern works. This is different from observing superficial features. Cause and effect are given. Such claims are subject to empirical evidence of falsification.

Only considering the superficial properties of a pattern is a dangerous path because developers may not understand the consequences of their design decisions. And they may be unaware of alternative design approaches or other plausible heuristics.

A pattern can be represented only indirectly; a pattern is the emergent wholeness that is common to all its exemplars. It cannot be found in one single example, however it is manifested in each example.

A good pattern description includes a model as one representative instance of the pattern.

Software pattern solutions are depicted in isolation from any realistic context.

To be useful, the information provided in a pattern description needs to fit prior knowledge and preferences of the target audience.

The patterns are out there, yes. But we need to understand that we only know parts of the whole story. Our ways of observing and analyzing good designs are limited. Never assume that you know everything about a pattern. There is always more to it.

The best we can do is to tell a story about the patterns. Such a story unfolds the inner relations of the wholeness of a good design. However, stories are never complete. Each stories has holes. Sometimes we are cutting out facts for convenience or to make the parts fit.

A story can be re-told in many different ways, with many variations, different levels of details, and commentary. A story can also develop. Parts can change. New facts emerge. Elements that become obvious over time can be left out.