Are Software Patterns Simply a Handy Way to Package Design Heuristics?

REBECCA WIRFS-BROCK, Wirfs-Brock Associates

Billy Vaughn Koen, in *Discussion of the Method: Conducting the Engineer's Approach to Problem Solving,* defines a heuristic as anything that provides a plausible direction in the solution of a problem, but in the final analysis is unjustified, incapable of justification, and potentially fallible. Software patterns might be considered nicely packaged heuristics in that they provide a context for the problem, and offer plausible solutions along with forces that the designer needs to consider when implementing a solution. Like any heuristic, software patterns come with no guarantees that they will solve the current problem at hand. A dedicated group of authors in the patterns community continues to write patterns, collections of patterns, and more ambitiously weave patterns into pattern languages that attempt to cover paths to solutions in a particular problem space. Are we deluding ourselves about the utility of these efforts? Or is there something important about both the form and use of patterns in the larger context of design heuristics that we need to understand?

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. Are Software Patterns Simply a Handy Way to Package Design Heuristics? 24th Conference on Pattern Languages of Programming (PLoP), PLoP 2017, Oct 23-25 2017, 15 pages.

1. INTRODUCTION

We have been writing patterns for over twenty years. The patterns community has branched from its initial software patterns roots, to write patterns encompassing many areas of human endeavor including, but not limited to, human interactions and collaborations, project management, software development processes, organization design, change, leadership, collaborative endeavors, beauty, teaching, pedagogy and learning. While there is evidence that software design and architecture patterns have successfully been applied in different contexts [Hohp], my aspiration is for these patterns, and software design heuristics in general, to have a much broader impact.

So what will it take specifically for software patterns to become more widely known, shared and used along with other software design heuristics and practices? Are there things we can learn about the nature of patterns, how they are described, and how they are understood, chosen and then applied that can improve our ability to communicate our patterns?

Inspired by Billy Vaughn Koen's philosophy of engineering heuristics, as explained in his *Discussion of the Method: Conducting the Engineer's Approach to Problem Solving* [Koen], this essay explores some characteristics of patterns, forms connections between patterns and Vaughn Koen heuristics, and lays out some challenges (and frustrations) in using both skillfully.

2. THE CONNECTION BETWEEN HEURISTICS AND PATTERNS

In *Discussion of the Method,* Billy Vaughn Koen defines a heuristic as, "anything that provides a plausible aid or direction in the solution of a problem but is in the final analysis unjustified, incapable of justification, and potentially fallible." Engineers try to create practical solutions to problems that are not fully understood. If you desire to create or change a system (whether social, political, physical or otherwise), opting for what you consider to be the best available heuristics to apply as you balance conflicting or poorly understood criteria for success, then you are solving an engineering problem. Rarely are engineering problems well defined. Instead, we determine what the actual problem is based on diffuse, changing requirements. To solve that problem, we successively apply heuristics based on our imperfect knowledge of both the current situation as well as the outcome of taking any specific action. Yet we press onward, applying heuristics because we believe our actions will likely result in positive changes.

A preliminary version of this paper was presented in a writers' workshop at the 24th Conference on Pattern Languages of Programs (PLoP). PLoP'17, OCTOBER 23-25, Vancouver, BC, Canada. HILLSIDE 978-1-941652-06-0

According to Koen, heuristics have these distinguishing characteristics:

1. A heuristic does not guarantee a solution
2. A heuristic may contradict other heuristics
3. A heuristic reduces the search time for solving a problem
4. The acceptance (or applicability) of a heuristic depends on the immediate context instead of an absolute standard.

Heuristics offer plausible approaches to solving problems, not infallible ones. There is always a chance that applying a heuristic won't move you closer to your goal. Yet even though heuristics can fail, they are readily applied to solve complex problems where the nature of the problem or the desired characteristics of a solution (or even a plausible direction forward) may not be clear. You choose a heuristic if it seems to fit the situation at hand, rather than asking how good is it compared against some gold standard. This is contrast to the scientific standard of, "is it consistent with the assumed truth as we currently know it?"

It is worth a short digression to contrast scientific theories with heuristics. If two scientific theories predict different answers to a question, the scientific method to resolve this conflict is to devise experiments that conclusively show that one theory is better because it explains more of what is observed than the alternative, competing theory. Any new, better theory needs to be consistent with pre-existing experimental results and in general, at least as accurate in its predictions as any pre-existing theory. Scientific theory building seeks to weed out lesser theories and supplant them with more elegant, simpler, or encompassing explanations. Science seeks single, better truths. In a Darwinian world of scientific theories, one theory wins out against another because it is better at describing and predicting observable outcomes. Einstein's theory of relativity replaced Newton's laws as a more encompassing scientific truth. Yet in engineering problem solving we still use Newton's law of gravitation because it quickly yields an excellent approximation of the effects of gravity when dealing with smallish masses travelling at sub-light speed. General relativity is required only when there is a need for extreme precision; when dealing with strong gravitational fields, such as those found near extremely massive dense objects; or at very close distances such as Mercury's orbit around the sun. Newton's law still works; but as a pretty useful heuristic, not an absolute law.

Heuristics are all about context, not conflict. If a heuristic is useful and expedient, by all means pick it up and use it! As noted, Newton's law of gravitation works for most practical engineering applications. However, heuristics are not timeless. When we had to really worry about memory footprint, software design heuristics for overlaying code and creating efficient data structures were extremely important. While they still may be useful in certain contexts, new memory organizations and multi-core processors have created different design concerns. Heuristics may fade from disuse because they are no longer so useful.

But software design heuristics may simply fade from use (even if still useful) because they are no longer in fashion. And technology drives fashion. If micro services and cloud-based computing are in, new patterns written about them will push aside older *Pattern-Oriented Software Architecture* [Busc] patterns.

So how are heuristics and patterns related? Are patterns simply a particular formulation of a heuristic, or is there something more to patterns and patterns languages?

Patterns describe and characterize potential solutions to common or recurring problems. Christopher Alexander in *Notes on the Synthesis of Form* [Alex] cautions that, "in the case of a real design problem, even our conviction that there is such a thing as fit [of solution to problem] to be achieved is curiously flimsy and insubstantial. We are searching for some kind of harmony between two intangibles: a form which we have not yet designed, and a context which we cannot properly describe."

Patterns, pattern collections, and pattern languages are our attempt at communicating nuances of problems and plausible approaches to solving them. What is unique about software patterns is that they don't just tell a designer what to do. In addition, they can provide a rich context that explains under what situations the pattern has been found to be useful, tradeoffs to consider, and an outline of a potential solution and consequences of applying that solution.

Software patterns are neatly "packaged" heuristics that offer design wisdom along with plausible steps forward. Software pattern languages are even more ambitious. They group and relate patterns in a particular problem domain, showing not only how individual patterns relate, but how a larger problem can be tackled by decomposing it into smaller sub-problems-with-potential-solutions, each written as individual patterns.

Restating Vaughn Koen's heuristic characteristics in terms of patterns, we get these statements about patterns:

1. ~~A heuristic~~ The application of a pattern does not guarantee a solution
2. A ~~heuristic~~ pattern may contradict or compete with other patterns ~~heuristics~~

3. A ~~heuristic~~ pattern, pattern collection, or pattern language reduces the search time for solving a problem

4. The acceptance (or applicability) of a ~~heuristic~~ pattern depends on the immediate context instead of an absolute standard.

Let's see how these characterizations apply to software patterns, patterns collections and pattern languages.

2.1 The application of any pattern does not guarantee a solution.

Certain pattern forms, in particular the early software design patterns described in *Design Patterns: Elements of Reusable Object-Oriented* [Gamma] emphasized evidence of their patterns' utility. They cited references to proven implementations that used their patterns. In the early days of pattern writing, Jim Coplien coined the "rule of three" [Copl]: If you couldn't find three distinct instances of a potential pattern's use, it is not worthy of being considered a pattern. This "rule" may have helped weed out patterns that weren't utilitarian enough (and originally, it was intended to weed out academic research that hadn't been put into practice), but the fact remains: using a pattern won't guarantee a solution to your specific problem.

Yet does simply having solved a problem three times using a similar approach mean that it is worthy of being a pattern? Or a useful heuristic? When does a potential solution to a recurring problem cross into the realm of being a broadly useful solution, whether guaranteed or not?

As a pattern author, I tend to conservatively offer up my patterns, taking care not to over- or undersell their utility. I am not alone in this.

This tentativeness seems to be part of our pattern writing culture. I've been to many PLoP writers' workshops where we spend a great deal of time trying to line up forces with solutions (and try to come up with both negative and positive consequences of applying a pattern). There's both an upside and a downside to using any pattern. We know that as software makers. We pattern writers are a conservative bunch.

Some pattern authors include cautions and conditions and consequences of using their patterns. Some also include alternative solutions and variations. As a nuanced, informed pattern reader, I like to know my options. Because I know patterns are not foolproof, I find myself drawn to pattern forms that contain descriptions full of subtleties, alternatives, and caveats.

Readers of these richer pattern texts, those filled with forces and considerations and solution variants and extra discussions, have to engage deeply with the pattern before they can determine whether it fits their specific problem.

In the early days of patterns, I remember people forming study groups to fully appreciate the GOF patterns[1].

Rich pattern forms force readers to think deeply and come to their own convictions and conclusions about whether that particular pattern is useful. It is almost a rite of passage: if you can read and comprehend all these things about this pattern, maybe then you are worthy of using it.

Pattern readers looking for quick fixes to their problems or ready-at-hand solutions can be frustrated by the effort it takes to understand, yet alone apply such patterns.

Yet, as a learner of a new pattern, I may want to know its pedigree. Wow! It has been used in all these situations. I could use it, too. Still I need to see rather quickly how I can apply it to my problem.

In an unscientific study, at the time I wrote this paragraph (July 12, 2017), I compared the Amazon rankings for *Head First Design Patterns: A Brain-Friendly Guide* (a fun-packed, lightweight way to learn the 23 original design patterns) with *Design Patterns: Elements of Reusable Object-Oriented Software* (the original source of those patterns). *Head First's* overall ranking in all books: 3,851; *Design Patterns'* ranking: 10,481. Even more telling, *Head First* is ranked 1st in the system analysis and design, object-oriented design, and object-oriented software design categories, while *Design Patterns* is 1st in software reuse and 5th in object-oriented software design. Perhaps inexperienced designers today (or their instructors) prefer more straightforward, simple writing or want fun, engaging ways to learn.

I suspect an even larger potential audience finds them online. More experienced developers who also have the time and the inclination to read detailed information, might prefer those original patterns. But not when they are seeking easy-to-digest advice for solving their current design problem. They may be impatient with details. They may not be looking for options. And design nuances, especially for an unfamiliar topic, will be lost on them.

It takes confidence to read, comprehend, and critique detailed design patterns. I used to teach object-design courses to professional software developers when object technology was relatively new. As part of the course I

[1] The authors of *Design Patterns* are sometimes referred to as the Gang of Four or GoF.

gave my students an in-class assignment of reading and interpreting a specific *GoF* pattern. They worked in small teams reading, and then discussing their assigned pattern. I asked that they relate it to their own work experience, and then share and teach the pattern to the rest of the class. Once they saw that I really did want them to critique both the writing and the pattern's solution, and that it was OK to be frustrated by a pattern, they really got into the exercise. Some shared alternative solutions they had seen or implemented. They came to realize that patterns don't present every plausible solution to a problem; only those solutions deemed reasonable given the pattern authors' experiences. My students might not have learned all the *GoF* patterns, but they gained an appreciation for how to approach, engage with, and comprehend detailed pattern descriptions.

2.2 A pattern may contradict or compete with other patterns

My goal as a software designer is to come up with practical solutions. This is a rough, uneven process. I wish it were more systematic and easily explained; but it is not. I don't start right away sorting through patterns and pattern collections for potential solutions. Instead, I'll take a look into various technologies, frameworks and platforms, and current industry trends (which are rarely described as patterns). More likely, unless I am a solo designer, others have previously made technology choices that already constrain my software design solutions.

That said, when it comes to choosing high-level ways of structuring my software, I do have in mind many higher-level structuring patterns. I consider these along with general rules-of-thumb, guidelines, or preferences I have accumulated over 30 years of making design choices. I am fully aware of several competing alternatives.

Let's look at a high-level architectural structuring example to illustrate my thinking. Although this architectural style is somewhat dated, it is still a useful example. In *Patterns of Enterprise Application Design* [Fowl02], Fowler identifies alternative patterns for structuring domain logic in a business application:

- A *Transaction Script* organizes business logic for a single business transaction into a procedure, which makes calls directly to the database or through a thin database wrapper.
- A *Table Module* organizes domain logic quite differently, into one class per database table. A single instance of a class contains functions that operate on elements in the table.
- A *Domain Model* organizes business logic into domain entities, value objects and services, each object representing some meaningful domain concept or behavior.

With both the Domain Model or Table Module approach, a *Service Layer* provides access to business functionality, controls transactions, and coordinates responses to/from either domain or table module objects (See Figure 1).

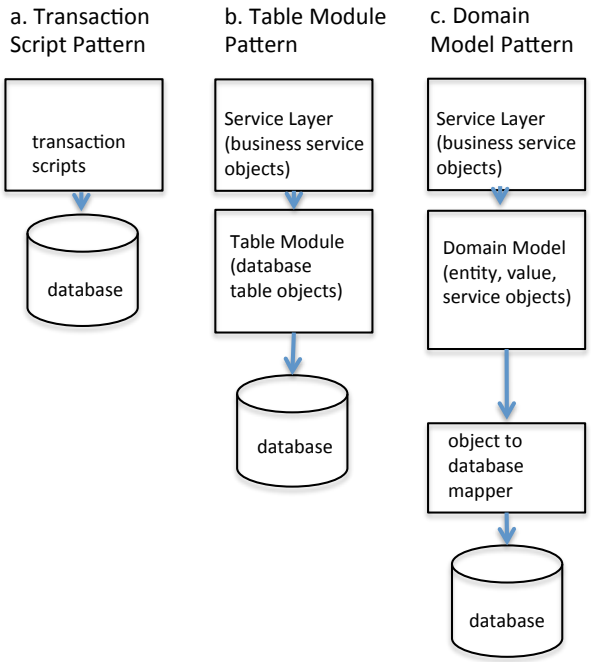

Figure 1. Three patterns for structuring enterprise applications: a. Transcription Script, Domain Model, and Table Module.

In the case of the *Transaction Script*, there is no concerted design effort to separate behaviors into distinct objects or to create an object model of the domain. With a *Table Module* solution, a designer doesn't worry about finding domain abstractions, per se, but acknowledges that the database schema design is best used as is. Typically, frameworks provide mechanisms for specifying classes which represent either an entire database table or projections (views) onto the database.

In the *Domain Model*, manipulation of data takes place by "model" objects designed to interact to accomplish work and to store and retrieve data. A single class represents a single instance of a domain entity, which if persistent, is retrieved and stored in a database. Low-level database details can be "hidden" from a more abstract view of that information embodied in domain objects (which comes with both design benefits and drawbacks). Furthermore, there is another choice; where to locate complex domain behaviors:

- In an *anemic domain model*, complex logic is located outside of domain entities; domain entities are designed primarily to be cohesive containers of data.
- In a *rich domain model*, business logic is considered intrinsic to the domain and is consequently located either in domain entities or as separate domain-related service objects.

Because I've seen so many workable solutions, I hesitate to say any one is "best." Still, some are aesthetically more pleasing to me.

Even though I know of different ways of structuring an enterprise application, I have my own style and distinct preferences because of my background, that is, the collected set of heuristics and experiences I have assimilated. I have built up a toolkit of heuristics, both consciously and unconsciously, for structuring enterprise applications through writing code, reviewing others' code and designs, using various frameworks, learning about object modeling and analysis in general, and becoming familiar with many patterns.

Coming from a Smalltalk programming background, where everything is an object, I view interacting networks of objects as good, workable ways of structuring many design solutions. Because of my object design roots and invention of the Responsibility-Driven Design method [Wirf90, Wirf02], I know that any object is capable of both knowing and doing things.

I'm primed to prefer *Domain Models* to *Table Modules* or *Transaction Scripts*, and rich domain models to anemic ones.

In fact, when seeking solutions to an enterprise application design, I may *only* see as viable options the choice between how rich or anemic my *Domain Model* should be. I quickly discount the *Table Module* approach because I view it as overly constraining my software objects' designs to align with the database schema.

And I wouldn't consider a transaction scripting approach to construct complex enterprise systems because it quickly becomes unmanageable. And I am happy to also discard it as a viable solution for even relatively small applications.

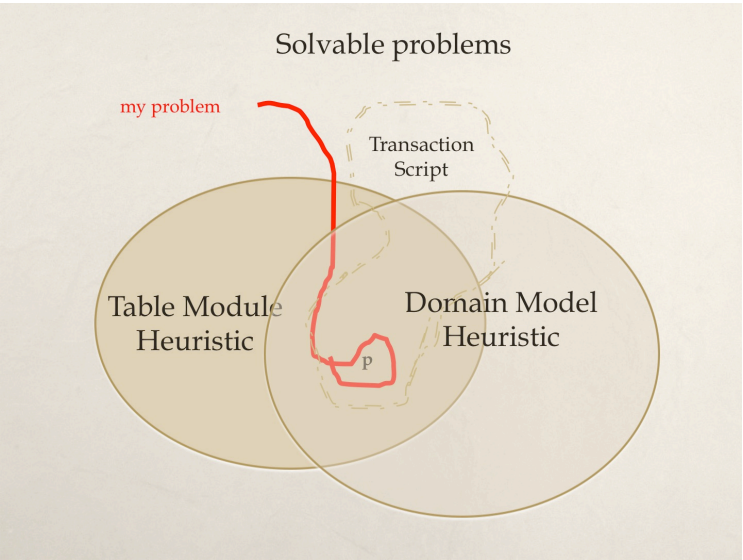

Figure 2. Transaction Scripts and Table Modules fade from my consideration because of my preference for Domain Model solutions

Yet given the above descriptions, it is clear that the *Domain Model* pattern contains more discrete blocks of functionality than the transaction script. So arguably, it appears like a more complex solution. So what are justifiable reasons for introducing this additional complexity into my design?

I prefer domain models because I have found them to be useful in many contexts and am comfortable with that style. Domain models are easy for me to invent. That doesn't mean I won't ever solve a design problem using a Transaction Script...it just isn't the first or second or third heuristic that comes to mind. It simply fades from my solution search space, because my positive experiences with other approaches are so much stronger. I find myself prefering a *Domain Model* solution over a *Table Module* in many situations because I don't want to have my application behaviors coupled and constrained to work awkwardly with an existing database schema, which is likely not well-matched to my application's desired behavior. I accept the complexities of an object-to-database mapper in order to remove this design constraint.

2.3 A Pattern, Pattern Collection, or Pattern Language Reduces the Solution Search Time

It is only at the beginning of a design project do I use domain model or table module or transaction scripts because they establish the overall high-level structure of my design solution. And then, the ongoing more detailed design work begins. Knowing large-grained application-structuring patterns is also useful for characterizing existing designs. What is most important for solving day-to-day design problems is knowledge of a wide variety of lower level, more detailed patterns along with both general and more specific design heuristics.

And yet, I have different degrees of familiarity with lower-level patterns.

I remember each GoF pattern because they were among the first software patterns. I reviewed a pre-publication version of their book. I really studied those patterns. New and entirely novel, they were burned into my brain. The authors organized 23 patterns into three categories: Creational, Structural, and Behavioral. Perhaps, for some, this organization helped reduce search time. But given that I think of objects as encapsulating both data and behavior…well, those categories haven't proven useful to distinguish between the various patterns. Consequently, I largely ignore these categories. Adapters, Bridges, Facades enable integration and encapsulation of disparate parts of systems. Are they structural or behavioral patterns? Since they organize parts of the system, I'm guessing they are structural. But I'm not sure unless I double check. Ah, yes, they are. Good. So what?

Factories allow me to create objects indirectly, reducing the dependency on specific class names. Ah, obviously a creational pattern.

Strategies, State and Visitor patterns define classes that perform variable actions. Certainly these are behavioral patterns. But every object is capable of having behavior.

Mementos allow me to store and retrieve objects without breaking encapsulation. Seems like both behavioral and structural objects to me. But that cannot be. A pattern, in this collection at least, fits into only one category.

As a co-author of a recent patterns collection [Yode14a, Yode14b, Yode14c, Yode15, Yode16a, Yode16b] I recollect spending time trying to identify relevant categories with my co-authors, at the prompting of our shepherds and writing workshop members. As it was, we discovered through short activities at various patterns conferences and workshops, that our pattern readers were cleverer at organizing our patterns than we were. Not surprisingly, there are several equally valid categorizations.

Pattern authors may think they need to place individual patterns into distinct categories so as to reduce a designer's cognitive load (and perhaps their solution search time). But I'm not so sure about this. Perhaps, instead of categorizing our patterns we should characterize, that is, tag them with multiple characteristics, and let these characterizations emerge as we build our collections and share them with others.

I recently listened to Ralph Johnson's 2014 SugarLoaf PLoP keynote on Twenty Years of Software Patterns (https://www.youtube.com/watch?v=ALxQdnOdYXQ). In this talk, Johnson proposed a more effective way of categorizing the GoF patterns (core, creational, and peripheral) and shared that at the time they wrote their book, they found the categories to be rather dubious, but went with them for lack of any better scheme. Aha! My hunch about these categories was finally confirmed.

In his talk, Johnson also introduced additional patterns that he felt were fundamental to good object-oriented programs, but were missing in their initial work. This included, among others, Value Object and Null Object, and Dependency Injection. I've known these patterns for a while (having learned about them from other authors and actually applying them). And because I have used them, it doesn't matter to me whether they are part of any particular collection. They are still part of my heuristic toolkit.

Richard Gabriel likens a patterns collection to a parts store. If you are looking for a particular software heuristic to apply, you go to the aisle where all those kinds of parts are. The size of a pattern collection is a factor in my remembering it, as well as the granularity of the individual patterns, how "closely" they are related, and how closely they match my current problem. Knowing what the general shape of what you are searching for is one thing; knowing that there are always "gaps" in any written pattern collection is equally important. And then? You unconsciously fill those gaps with things you've learned from experience.

Johnson also discussed common misuses and/or misimpressions of some "dangerous" patterns (most notably Singleton and Composite). In response to a question about missing gaps in the creational patterns, Johnson gave a clear explanation of the nuances of several (as yet undocumented) creational patterns and hinted at how they might be organized. Authors only write patterns for what they know. Pattern authors can't anticipate what future designers encounter. Not only are patterns potentially fallible, patterns collections are incomplete. And, to stay relevant, they need refreshing, based on current experience and practices.

I expect to fill gaps in any particular patterns collection with my own experiences. Over time, I have learned how to knit together and reconcile patterns from diverse sources. As I design, I additionally apply both general and specific design heuristics to guide me whether or not they are written as patterns.

The more patterns and pattern collections there are, and the more disjointed they are, the harder it becomes to locate just what I need. Even more daunting is the task of searching through multiple, partly overlapping collections, seeking what might be the next best design heuristic to apply. I don't have a handy browser or tool that aids my search.

Consequently, I need to have mentally organized and loosely arranged the patterns collections I do know of so that I can call them to mind, when need be. I trust my instincts and pay attention to what aspects of my design emerge as important to me as I go.

I don't remember many patterns collections in their entirety.

With later patterns collections I encountered, I tended to learn of them in general lumps (not necessarily remembering each individual pattern, but instead trying to understand the general design principles behind them and the gist of their concerns).

For example, Fowler's *Analysis Patterns* [Fowl96] are a loosely related collection heuristics for modeling various aspects of specific domains that Fowler collected over a period of time. With 50 patterns in that collection, I have to search it carefully (actually search, because I haven't committed each pattern to memory) to find whether any particular pattern fits my needs.

All pattern collections have gaps and warts and differences in pattern granularities as well as more or less useful patterns. Because I have assimilated many such collections, these inconsistencies don't trip me up. I suspect a less experienced designer might have a great deal more difficulty.

For example, I pretty much discount the GoF Singleton, Flyweight, and Iterator patterns. I know of them because those first 23 patterns have been indelibly burned into my memory. They are there at ready recall, even though I have never used them to solve any design problem (perhaps this mental clutter leaves less room in my brain to distinguish new patterns I might shove in there).

But many times I don't need to use GoF patterns at all...I simply make design choices about what behaviors and data to package together. And to do so, I fall back on my personal set of design heuristics for encapsulating variability and supporting design flexibility.

A few have been written as patterns; most have not. But I have written and spoken to others about most of them. Using Responsibility-Driven Design principles, heuristics, and design characterizations I can figure out how to tackle new design problems as well as decipher existing designs. For example, from Responsibility-Driven Design [Wirf02] I use these basic role stereotypes to characterize objects:

Information holders—that know and provide information.
Structurers—that maintain relationships between objects.
Service providers—that performs work on demand.
Coordinators—that react to events by delegating to others.
Controllers—which make decisions & direct actions.
Interfacers—which transform information and requests between distinct parts of a software system.

These characterizations help me understand more deeply the roles of design elements that are part of my current design as well as those classes or objects described by any software design pattern. Along with the

Responsibility-Driven Design heuristic, "Make objects smarter by blending stereotypes" they allow me to construct:

- information holders that compute or derive information based on other information they maintain;
- service providers that maintain information to be more efficient;
- structurers that answer deeper questions or derive facts about what they are structuring; and
- interfacers that also transform;

without having to search for any specific pattern to apply. I apply object role characterizations, along with the knowledge of various heuristics and design tradeoffs, to guide how I structure groups of collaborating objects (I also characterize control styles, identify trust regions, and am familiar with different error recovery and fault handling strategies). All these design characterizations and heuristics are part of my working design vocabulary and ways of thinking about a design.

Heuristics and patterns and ways of seeing and understanding the design space are passed along to others by implementation examples, books, writing, teaching, and word of mouth. If someone doesn't spread them, keep them relevant, and promote them, they will fade from fashion (not because they aren't useful, but because they become less known).

From *Streamlined Object Modeling: Patterns, Rules, and Implementation*[2] [Nico], which is not well-known today, I learned the useful distinction between descriptive, time-dependent, lifecycle state, and operational state attributes), which broadly is useful in domain modeling as I apply this heuristic:

Heuristic: Characterize a domain entity's attributes to understand/find/identify needed system behaviors.

Because I know of *Streamlined Object Modeling* attribute characterizations, I can identify and better understand design domain behaviors in an emerging domain model as I apply Domain-Driven Design patterns [Evans].

In *Streamlined Object Modeling* there are patterns for modeling people, places and things, which I have boiled down to three simple heuristics shown as shown in Figure 3.

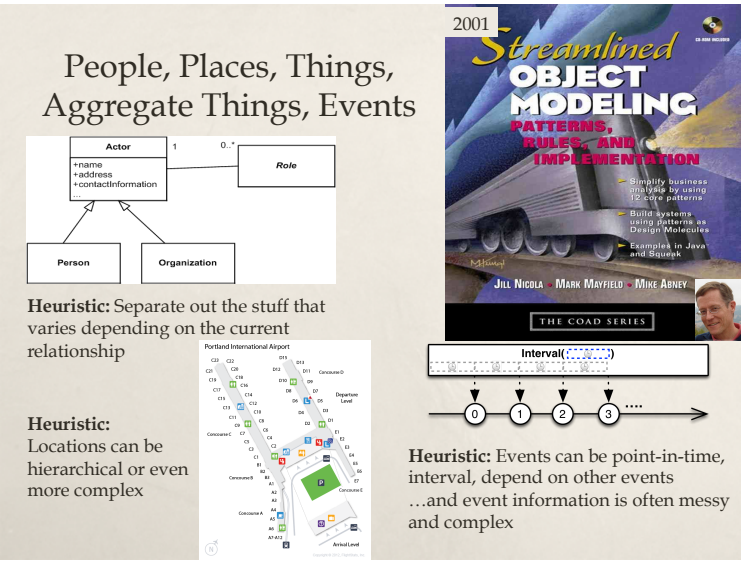

Figure 3: Patterns for People, Places, and Events restated as simple heuristics

Knowing the interconnections and linkages, whether stated or not, between various pattern collections is equally important. The details of the patterns in these two particular catalogs don't seem as important to remember as does the formulations of more fundamental heuristics for structuring and characterizing entities and relationships between them. If I distill the essence of several patterns into higher-level, more abstract heuristics it reduces my mental clutter and seems to enrich the quality of my design thinking.

[2] Rules as meant in the title of this book are roughly equivalent in meaning to Vaughn Koen's definition of heuristics.

2.4 The acceptance (or applicability) of a pattern depends on the immediate context instead of an absolute standard

Software patterns that describe their context for use that I can also quickly match with my own needs, offer an advantage over those that either leave out or make assumptions about the context. Often confounding to pattern newcomers is the fact that their solutions to real-world problems are always more complex than the stylized ones written about in any particular software pattern.

I have used and extended several patterns from different collections on more than one occasion. I remember feeling pleased to find that my solutions were more complex and nuanced than the patterns described in these books and that I had found a clever way to extend and augment those patterns. (A solution that I worked on that represented complex roles and privileges for individuals belonging to multiple organizational structures far exceeded the simple relationships in the Accountability and Accounting patterns described in Fowler's *Analysis Patterns*). But I also remember the discomfort of my less pattern savvy colleagues who felt that they hadn't "gotten" a pattern correctly if we needed to refine or extend it. Only after reviewing our solutions with Martin Fowler and passing along to my colleagues confirmation that indeed, he thought our problem seemed to warrant a more complex solution, did they feel comfortable with our design.

Gigerenzer, in *Simple Heuristics that Make us Smart* [Gig], says that useful simple heuristics apply to specific environments and tasks, but do not contain enough detail to match any one environment precisely. While patterns contain more information and specifics than most of the fast-and-frugal heuristics described by Gigerenzer, this seems like an ideal tactic that pattern authors should ascribe to. The goal of our writing should be to neither over- or under- specify the patterns' context and forces that must be balanced. Software patterns that don't provide much context, or offer too broad or too narrow a context, can be difficult to learn and apply. In what context are GOF patterns useful? The authors claim that their design patterns were useful for designing object-oriented systems intended to be reusable and extensible. Some argue that several patterns in the collection have narrow scope and limited use. Singleton and Iterator are notable for being held in disregard.

But the notion of a Façade or an Adapter is a useful construct, regardless of implementation technology. What is missing from this specific object-oriented pattern collection is the more general heuristic: encapsulate what you don't want to expose to the rest of the system.

Limiting GoF patterns' utility to only an object-oriented programming solution is, indeed, limiting.

Expanding on them so they can be applied more generally is something I've done, over time, as I've built up my design repertoire. I've broadened their context through experience.

In the *Pattern-Oriented Software Architecture* collection [Busc], a context section follows immediately after the pattern summary and a short example. In it, the *Command Processor Pattern* separates the request for a service from its execution. It has this context: "Applications that need flexible and extensible user interfaces, or applications that provide services related to the execution of user functions, such as scheduling or undo."

As I was writing this essay, I was struck by how dated that context seemed.

Encapsulated actions, or services that can be invoked, are useful in much broader contexts than user-system interactions. These days is a commonly accepted software design practice to separate requests from their execution, regardless of whether there is the need to undo or redo an action. And more recently, the Command pattern has become part of the CQRS *Command-Query-Responsibility-Segregation* pattern [https://martinfowler.com/bliki/CQRS.html].

Not only does the applicability of a pattern rely on the immediate problem's context, but, as our software design experiences evolve, the appropriate context for any particular pattern might also reasonably be expected to shift, expand, or contract. We software designers are clever problem solvers. We take ideas and adapt them as we see fit. Our solutions embody patterns or ideas based on written patterns, even if we do not know or remember their names.

3. SOME CHALLENGES WITH SOFTWARE PATTERNS AND DESIGN HEURISTICS

However, if we want to keep written software patterns alive and relevant, there is an urgent need for us to broaden, refine, and refresh existing patterns' contexts and their applicability. We also need to distinguish between more general design heuristics, larger grained software patterns for structuring systems, finer-grained, more specialized patterns, patterns for specialized implementations and architectures, and patterns that are more or less useful.

3.1 Too many heuristics are overwhelming

In 1997, Arthur Riel published *Object-Oriented Design Heuristics* [Riel], an exposition of 60 heuristics for designing object-oriented software. The heuristics in his collection were organized into these categories: classes and objects, the topology of classes, the relationship between classes and objects, inheritance (as well as multiple inheritance), and associations. Each heuristics was codified in a single sentence and explained in more detail. I didn't recall Riel's work until I revised this paper. As I skimmed his book, once again I found myself vigorously arguing against several of his heuristics, looking to take exception.

The very first class heuristic (Heuristic 2.1) is, "All data should be hidden within its class." No public variables. Another heuristic (Heuristic 3.2): "Do not create god classes/objects in your system. Be very suspicious of a class whose name contains Driver, Manager, System, or Subsystem."

I don't respond well to absolutes.

I remember challenging myself when I first read his book to restate some of though heuristics in ways that I could find more palatable. Some I simply couldn't reconcile with my existing design practices, so I quickly discounted them. Some were too vague to be useful. A heuristic should guide us to make a decision and then take action. But what concrete action could I take with Heuristic 3.7, "Eliminate irrelevant classes in your design?" And I wasn't certain about the practicality of Heuristic 4.6, either: "Most of the methods defined on a class should be using most of the data members most of the time." Based on my experience, I found that different behaviors relied on different data encapsulated within an object. So it seemed quite reasonable to me that some methods would use certain data members, while other methods would use other subsets of the data members. What are some reasonable exceptions to this "generalized" data member access heuristic? What was the point of trying to enforce the heuristic that most of the data members should be used most of the time? Was it to tease out behaviors that belonged in different objects? If so, there certainly are better ways of expressing rules-of-thumb for refactoring objects and behaviors.

Chapter 5 of Riel's book has a collection of 19 heuristics for Inheritance. I thought he started out reasonably with Heuristic 5.1, "Inheritance should be used only to model a specialization hierarchy" and Heuristic 5.2, "Derived classes must have knowledge of their base class by definition, but base classes should not know anything about their derived classes." But then, it seemed he quickly got into more controversial territory with these:

> **Heuristic 5.3** All data in a base class should be private; do not use protected data.
> **Heuristic 5.4** In theory inheritance hierarchies should be deep—the deeper, the better
> **Heuristic 5.5** In practice, inheritance hierarchies should be no deeper than an average person can keep in his or her short-term memory. A popular value for this depth is six.

I could go into detail about the many problems I have accepting those three heuristics. But I won't except to state that heuristics that contain magic numbers related to short term-memory size and absolutes interspersed with "shoulds" cause me to fight against instead of seeking to understand the reasoning behind them.

At the time, Riel's heuristics considered en masse seemed far too much work to sort out and reconcile with my own design practices. That is one reason I suspect why I didn't remember them.

Still, that book contains some heuristics I think are reasonable. Heuristics that have stood up over time which others have been echoed and refined. One example, Heuristic 2.8: "A class should capture one and only one key abstraction," is similar to the Single Responsibility Principle popularized by Robert Martin [Mart]. However, once again, I have found that Single Responsibility Principle to be commonly misinterpreted by students of object design to mean, "a class should only have a single method" instead of a class should have a singular or cohesive purpose.

Software pattern or design heuristic collections need to be mostly useful and at the right conceptual level, or they will be easily forgotten. Heuristics, without accompanying explanation and context, which fortunately most pattern descriptions provide, are open to misinterpretation. Heuristics boiled down to simple pithy phrases might be good reminders, but only if you understand the reasoning behind those phrases. That's where patterns offer a distinct advantage. They include enough details, context, and solution for you to sink your teeth into.

Learning too many heuristics (especially contradictory ones), without seeing workable, concrete examples or having varied experiences applying them, isn't helpful either. I only became proficient at object design through a combination of practice, learning about patterns and patterns collections, being around people who

were a lot much more clever than I am, as well having a propensity for distilling specific heuristics, phrasing them in my own words, and making my own observations on what seemed to work.

But learning software design patterns isn't enough to become a good designer. Sure, we need heuristics that can be broadly applied to construct workable design solutions. But we also need heuristics for comprehending the fundamental structures and behaviors in others' code as well. And we need heuristics for knowing what's important about our current design. And then, we need to know heuristics for understanding how to move to our design forward.

3.2 We don't know how designs actually evolve

Riel's book contained a tantalizing chapter on his view on the relationship between software patterns and design heuristics. Riel thought his heuristics were "a gateway through which a designer can move from a bad design pattern [e.g. a pattern that had been applied that didn't improve the design] to a good design pattern." Riel postulated that his style of heuristics could lead designers to consider making design changes that involved applying patterns, whether or not these patterns had yet been discovered. Although he gave a couple of simple examples, he left the exercise of generating more transformation pattern sequences (e.g. current design-> applied heuristic-> new pattern applied to improve the design) as a future research topic. Many of the Riel's heuristics are about detailed implementation concerns, for example the proper use of class variables instead of globals, or not overriding inherited methods with no-ops. Because these heuristics are more about stylized use of object-oriented programming language capabilities, I suspect it would be not very fruitful to hunt for the patterns found when "fixing" the design to conform to them.

In *Refactoring to Patterns* [Ker], Joshua Kerievsky, explores the motivations and reasons for a designer to move toward as well as away from the twenty-three basic GoF patterns during implementation. I liked reading Kerievsky's book because it got me inside his head as a designer. And yet, in his steps toward and away from "pure" implementation of patterns as originally sketched out by the GoF authors seemed to simply be realistic examples of how a designer adapts any pattern to their particular context.

When I design, I start with initial design ideas and then move in uneven steps to a more nuanced understanding of what aspects of my emerging solution are important. I don't move towards or away from specific design patterns. Instead, I move between general and more specific design heuristics as my design and implementation unfolds. It is only when there is some tension or nagging uncertainty or when I move into an unfamiliar area of design that I search for a handy pattern to apply. And then I continue on until I bump up against another design challenge that requires active search for heuristics that are not already ingrained into my overall design heuristic gestalt.

There is still much that to learn about how we software designers actually design.

Another book I looked back on as I was writing this essay was *Object-oriented Reengineering Patterns* by Serge Demeyer, Stéphane Ducasse, and Oscar Nierstrasz [Dem]. This book stands out as example of not just a single pattern language, but instead a well-organized set of related pattern languages. It contains pattern languages for understanding and improving on the design of existing object-oriented systems. Each chapter starts with a pattern map illustrating potential sequences of patterns based on activities you want to do next. For example, For example, Chapter 3 is about patterns for your first contact with a system (see Figure 4). This map illustrates the dynamic and iterative nature of applying these patterns. There is no obvious beginning or ending place, other than having a good-enough understanding of the system in order to make an initial assessment of the re-engineering effort. So you may cycle between talking with people about the system and verifying what you hear until you have enough information to proceed.

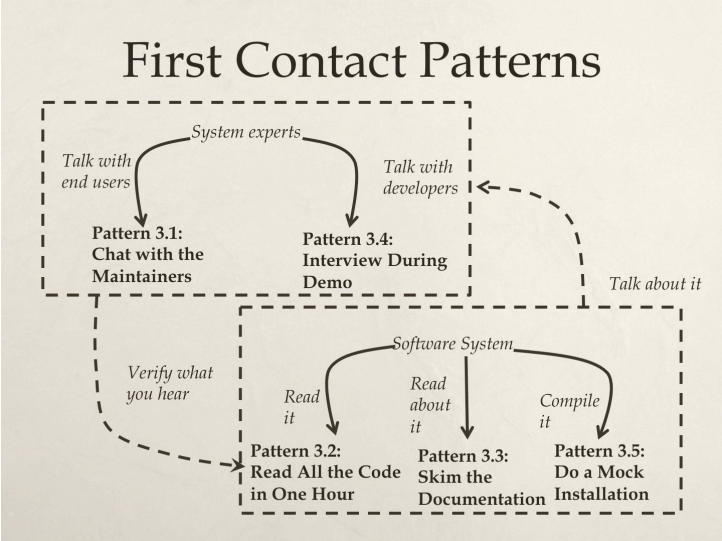

Figure 4. Patterns for First Contact with A System from *Object-Oriented Reengineering Patterns.*

Successive chapters address understanding the system more deeply, testing, migration strategies, nuts and bolts activities for deleting duplicated code, redistributing responsibilities, and even transforming conditionals to using polymorphism. Lower-level reengineering activities reference patterns written by others, as well they should. A pattern language need not be self-contained; it can leverage the work of others. In fact, it seems more expedient if it does.

3.3 The State of the Art (SOTA) Changes

We designers use the best tools, techniques and heuristics we have at hand. Looking through our current lens of experience, we may consider "legacy" software to be dated or quaint. Billy Vaughn Koen cautions us to not judge older designs too harshly. Our designs are based on heuristics we used at the time decisions were made. A problem that isn't unique to software, however, is that our software designs often outlive the lifespan originally envisioned for them. Consequently, software maintainers are stuck with sustaining software systems of enormous complexity that were constructed using perfectly reasonable heuristics. The only problem is that that may have been lost, forgotten, or if remembered, are considered arcane by current standards.

One challenge is to find practical ways to record the heuristics that were applied and our design rationale to inform our future selves and others. Decision records and design decision logs, while not new ideas, currently seem to be gaining some traction. An open source project on Github supports various forms of architecture decision records which can be checked in along with the code [Hend]. Our perspectives and collective heuristics change, grow, and evolve over time. In recognition of this reality, Michael Nygard proposes a format for decision records [Nyg] include decision status which can be "proposed", "accepted", "deprecated", or "superseded."

3.4 Heuristics compete with each other.

We designers come to our own conclusions about which heuristics should be part of our toolkit, and which ones we set aside. With experience and perspective, we may also come to appreciate that heuristics naturally compete with each other. Pattern authors could conceivably enhance our design decision-making, if they tagged patterns which are in direct competition with others as well as mentioning other approaches that they have seen, which they explicitly do not recommend, and consequently aren't worthy of being captured as a pattern. Patterns generally only tell us what to do instead of also what to not do (and why not to do something). This missing piece of the puzzle has to be filled in somehow, either through education or experience.

For example, *Domain Driven Design* offers three different ways to verify data passed into business applications. You can use the *Constraint pattern* and create constraint objects, which are responsible for verifying attributes. This is useful when verifying cross-attribute consistencies or attributes spread between

multiple objects. Or, you can define *Helper methods* directly invoked from the called method which receives the data. Or most simply, you can directly verify values passed into the called method.

Another equally viable alternative that isn't mentioned is to write specifications of syntactic constraints to be checked by framework code on the data before it is passed along to the application (a technique Java and JavaScript programmers are intimately familiar with). In all fairness these frameworks became popular shortly after the book was published.

As an exercise for students in my Enterprise Application Design course, I presented these various options, then had them justify why they might pick one approach over another. I then challenged them with increasingly complex situations to make them think more deeply about the limits to their designs.

If there are few syntactic checks for data, fine, go ahead and use pre-processing frameworks. If you want to meaningfully collect and display all errors in that data and inform the user of multiple errors, popular frameworks only go so far. But it is simple enough to build your own framework to do so (and I demonstrated a simple one that I had designed that did that). If validation rules can dynamically change based on the user or current application context, then helper methods or statically specified checks won't work. After throwing increasingly complex situations at my students, I advised them to determine what the current requirements are, and pick a preferred mechanism based on what they know. Don't anticipate changing requirements or the need for anticipated flexibility. When requirements change, design decisions need to be re-examined.

As a prudent designer, I try to pick the simplest mechanism that works reasonably well for most cases that I know of and then stick with it. I also value consistency over localized cleverness. But at the same time, I want to know what the tradeoffs are when I should ditch my preferred heuristic for a better alternative. Detailed comparisons of *Domain Driven Design* data verification as well as most other patterns are left to implementer or to the educator.

4. SUSTAINING SOFTWARE PATTERN PRACTICES

Patterns are neatly packaged heuristics that carry extra information valuable to the designer. While great stuff, they are also in danger of becoming lost, forgotten, or outdated. And they compete, along with many other design strategies and rules-of-thumb for our attention. So what can we do to breathe continued life into software patterns?

In *Pattern-Oriented Software Architecture* [Busc], the authors describe two early attempts at linking all the known software patterns. Effectively organizing the growing body of patterns was a problem even in the early days. They recount a pattern mapping exercise at the PLoP '95 conference, where authors wrote their patterns on paper, placed them on the floor, and then used string to link their patterns with other related ones: "A first picture of the pattern universe was thus drawn, although in a very informal, ad hoc and uncoordinated way. Nevertheless, about three hundred different patterns were connected this way." They tell of another attempt a few months later at organizing patterns at a Hillside Group retreat where attendees wrote over 150 brief pattern descriptions—so-called patlets—and then linked them using several relationships. They specifically pointed out two relationships between patterns: refinement, where one pattern is a special case of another, and contrasts with, where one pattern which appears at first glance to be similar to another is not.

In 1998, *The Patterns Handbook* [Ris98] was published. It collected, selected, and curated a number of definitive articles written by patterns experts in addition to examples, reviews, an annotated bibliography of published patterns along with contact information. Shortly thereafter, *The Pattern Almanac 2000* [Ris00] organized and briefly described 700 patterns. Two notable things strike me about the almanac. The categories identified by the pattern authors were used without modification. And there was no pattern critique. In hindsight this seems appropriate, given the relative newness of patterns. But even then, there was recognition that there was a problem that needed to be addressed. In the preface Linda Rising had this call to action: "I'd love to hear stories about patterns that have or have not worked for you, as well as insights regarding pattern evaluation and categorization."

Today, twenty years later, the software patterns ecosystem is much more complicated. The pattern landscape seems even more sprawling, disjointed, disorganized, and, at the same time, somewhat dated. While authors continue to churn out new patterns, valuable older patterns are being lost and forgotten. And only a few pattern authors and the communities that have formed around them are the ones keeping their collections fresh. As they do, they are creating islands of pattern knowledge, leaving other useful software patterns behind.

John Vlissides, in *Pattern Hatching* [Vlis], included a chapter on Seven Habits of Effective Pattern Writers. He observes, "A pattern, in contrast (to an integral) doesn't work in a vacuum. It provides the solution to just one problem, so it must cooperate with other patterns. Hence a pattern writer must contemplate not one

pattern but several, even some as yet unwritten…". He advises future pattern authors to take time to reflect, adhere to a structure, to be concrete early and often (and to include lots of real world examples), keep patterns distinct and complementary, present effectively, iterate tirelessly, and finally, to collect and incorporate feedback.

This seems like sound advice. But anticipating future patterns? Vlissides is asking a lot of us. Yet, to keep software patterns fresh we need to evolve and adapt them to fit our current software design practices. As our state of the art evolves, so too, should our patterns.

Additionally, to make sense of the larger body of software patterns as a whole, we need to show how patterns and collections relate and where they clash. There is a need to create and publish "maps" of overlapping/competing software design heuristics and patterns. And to recognize that not all patterns age gracefully. Patterns that do stay alive do so through continued use, but they also need ongoing critique, refinement, and improvement, whether that work is done by the original authors or by others [Wirf06].

I fear this larger sorting and revitalizing effort is a massive undertaking.

While there may be little incentive for authors of legacy patterns collections to refresh them, we as a larger software patterns and design community could become conservators of the more valuable software patterns. We'd need to take this on as a long-term effort. And as we do, there is also an opportunity to explore new and better ways of communicating software patterns and our other design heuristics to diverse readers and consumers.

For this revitalization effort to take hold, we need to somehow recreate a spirit of collaboration and enthusiasm that existed when both software patterns and Portland Pattern Repository wiki [Cunn] was new. We need to foster ongoing conversations and debate about patterns of the past, patterns of the present, and visions for software patterns in the future. We need to collect stories from patterns authors and experienced designers about how they actually design and how they weave patterns together with their other design heuristics. These stories might lead to deeper insights into how we actually design and be a source of inspiration for future design pattern languages.

5. ACKNOWLEDMENTS

I'd like to thank my shepherd, Mary Tedeschi, for reading early drafts of this essay and prompting me to come to some conclusions. I'd also like to thank my writers' workshop colleagues for giving me pointed advice about ways to make this essay more cohesive and compelling. Thanks particularly to Mary Lynn Manns, Helene Finidori, and Chris Richardson who made sure I heard what they had to say. And thanks to Allen Wirfs-Brock who paid attention to pesky details. This essay is stronger because of his critique and suggestions. And finally, I also would like to thank Richard Gabriel and Jenny Quillien for finding the weak spots in my writing and encouraging me to make stronger connections between patterns and heuristics.

REFERENCES

[Alex] Alexander, C. *Notes on the Synthesis of Form.* Harvard University Press, 1964.
[Busc] Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., Stal, M. *Pattern-Oriented Software Architecture: A System of Patterns.* Wiley, 1996.
[Copl] Coplien, J. The Rule of Three, http://wiki.c2.com/?RuleOfThree.
[Cunn] Cunningham,W. The Portland Pattern Repository, http://c2.com/ppr.
[Dem] Demeyer, S. Ducasse, S., Nierstrasz, O. *Object-oriented Reengineering Patterns* Morgan Kaufman, 2003.
[Evan] Evans, E. *Domain-Driven Design: Tackling Complexity in the Heart of Software.* Addison-Wesley, 2004.
[Fowl96] Fowler, M. *Analysis Patterns: Reusable Object Models.* Addison-Wesley, 1996.
[Fowl02] Fowler, M. *Patterns of Enterprise Application Software.* Addison-Wesley, 2002.
[Gamma] Gamma, E., Helm, R., Johnson, R., Vlissides, J. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.
[Gig] Gigerenzer, G., Tod, P. and the ABC Research Group. *Simple Heuristics That Make Us Smart.* Oxford University Press. 1999.
[Hend] Henderson, J. Architecture Decision Records, Github repository, https://github.com/joelparkerhenderson/architecture_decision_record
[Hohp] Hohpe, G., Wirfs-Brock, R., Yoder J., Zimmermann, O., "Twenty Years of Patterns' Impact," IEEE Software, Nov./Dec. 2013.
[Ker] Kerievsky, J. *Refactoring to Patterns.* Addison-Wesley, 2004.
[Koen] Koen, B.V. *Discussion of the method: Conducting the Engineer's approach to problem solving,* Oxford University Press, 2003.
[Mart] Martin, R. *Agile Software Development: Principles, Practices, and Practices.* Addison-Wesley, 2002.
[Nico] Nicola, J., Mayfield, M., Abney, M. *Streamlined Object Modeling: Patterns, Rules, and Implementation.* Prentice Hall, 2001.
[Nyg] Nygard, M. Blog post, "Documenting Architecture Decisions." http://thinkrelevance.com/blog/2011/11/15/documenting-architecture-decisions.
[Riel] Riel, A. *Object-Oriented Design Heuristics.* Addison-Wesley, 1996.
[Ris98] Rising, L. *The Patterns Handbook: Techniques, Strategies, and Applications.* SIGS, 1998.
[Ris00] Rising, L. *The Pattern Almanac.* Addison-Wesley, 2000.

[Wirf90] Wirfs-Brock, R., Wilkerson, B., Wiener, L. *Designing Object-Oriented Software.* Prentice Hall, 1990.
[Wirf02] Wirfs-Brock, R., McKean, A. *Object-Oriented Design: Roles, Responsibilities, and Collaborations.* Addison-Wesley, 2002.
[Wirf06] Wirfs-Brock, R.. "Refreshing Patterns" in the May/June 2006 issue of IEEE Software. Vol. 23, No. 3.
[Mart] Martin, R. *Agile Software Development, Principles, Patterns, and Practices,* Prentice Hall, 2003.
[Yode14a] Yoder, J. Wirfs-Brock, R., Aquilar, A. "QA to AQ: Patterns about transitioning from Quality Assurance to Agile Quality," AsianPLoP 2014.
[Yode14b] Yoder, J. Wirfs-Brock, R. "QA to AQ Part 2: Shifting from Quality Assurance to Agile Quality-Measuring and Monitoring Quality," PLoP 2014.
[Yode14c] Yoder, J. Wirfs-Brock, R., Washizaki, H. "QA to AQ Part 3: Shifting from Quality Assurance to Agile Quality-Tearing Down the Walls," SugarLoafPLoP 2014.
[Yode15] Yoder, J. Wirfs-Brock, R., Washizaki, H. "QA to AQ Part 4: Shifting from Quality Assurance to Agile Quality-Prioritizing Qualities and Making them Visible," PLoP 2015.
[Yode16a] Yoder, J. Wirfs-Brock, R., Washizaki, H. "QA to AQ Part 5: Being Agile At Quality-Growing Quality Awareness and Expertise," AsianPLoP 2016.
[Yode16b] Yoder, J. Wirfs-Brock, R., Washizaki, H. "QA to AQ Part 6: Being Agile At Quality-Enabling and Infusing Quality," PLoP 2016.
[Vlis] Vlissides, J. *Pattern Hatching: Design Patterns Applied.* Addison-Wesley, 1998.