Exploring the Generative Nature of Patterns

LISE HVATUM

REBECCA WIRFS-BROCK, Wirfs-Brock Associates

This is an about paper—one that we write to gain a better understanding, for ourselves and our readers alike, of the usefulness of a pattern language and what it takes for experience captured in pattern form to be of use to others. In it we look at what it means for individual patterns and collections to be generative and explore some differences between those that are highly generative and those that are not. We ask whether software design and software process patterns can be generative and, if so, what is their potential impact. Finally, we draw some conclusions on how these thoughts on pattern generativity could influence future writing of and support for generative patterns.

1. INTRODUCTION

Starting with EuroPLoP 2015, we published a number of papers on patterns for managing a software product backlog [HW2015, WH2016, HW2017, HW2018, WH2018, WH2019]. For a summary of our Magic Backlog patterns see Appendix A. A lot of work went into capturing, validating, reviewing, refining, and publishing the final papers—not only by us authors but by shepherds, workshop participants, and others supporting our efforts. At this time, we believe the collection to be mostly complete in that these patterns capture our detailed knowledge of creating and maintaining a healthy product backlog for software product development.

Having gotten this far, we should now be able to rest our case and enjoy a feeling of accomplishment. So why don't we? Because we struggle with key questions that remain unanswered. On the practical level, we ask ourselves if a team could take our work and use it to guide their efforts at creating their own backlog? Would our patterns help them be more effective? Or would they repeat the trials and errors we went through before attaining a good solution for their team? But even more, we wonder if the true goal of our patterns is the backlog itself or rather to improve the inner workings of the development team—that using these patterns leads them to a backlog that positively influences and enhances the way their team works.

There are several reasons why people write pattern papers and attend PLoP (Pattern Languages of Programs) conferences. It can be for academic achievements, to get feedback from a wider audience on company practices, or even because it is more rewarding to participate in a PLoP conference with a paper than without one.

But if your driving force behind writing patterns papers is to educate and help people work better, to enable others to create—whether it be microservice architectures, team practices, or sustainable communities—then you are probably among those who grapple with questions of how to best write patterns and patterns languages to make their contents accessible and impactful.

This leads us to seek a more thorough understanding of the qualities of patterns and to explore the potential they have to affect positive changes. In the early days of the patterns community there were deep discussions about these topics. We have researched the writings of Christopher Alexander as well as some founders of the software patterns movement. One fundamental characteristic of a pattern or a pattern language that they all agreed on as important is its generative qualities. We are especially interested in deepening our understanding of generativity and its implications on both pattern writing and pattern use.

---

2. WHY WE WRITE PATTERNS

Alva Noë, in *The Entanglement: How Art and Philosophy Make Us What We Are* [Noë], observes that, "Human beings are organized, in the large, and in the small, by habit, custom, technology, and biology. This organization is what lets us have a world and cope with it ... But it also constrains us; it holds us captive, defines our ordinary, and confines our intuitions." So how can we break out of our ruts?

Noë claims that art and philosophy, "are the ways that we re-organize ourselves. ... Art and philosophy require of us that we work ourselves over and make ourselves anew, individually and ensemble." Patterns aren't literary art or philosophy, but they share some important properties with them. Most important, patterns name and explain actions that previously we may have only dimly intuited. Patterns also articulate the essence of when it is appropriate to take certain actions and the potential effects of doing so.

We don't write patterns for pattern writing's sake, but to better understand a part of our world and the effects certain actions have on it. We want to convey that understanding to others. We want readers to be able apply our patterns, to reshape and creatively renew themselves and while doing so positively impact their world. In this regard. we are particularly inspired by the words of Devin Arrigo (Figure 1):

Humans are hard-wired to create.

There's something so satisfying about creating with your own two hands.

A treehouse. A poem. A song. A meal.

Human beings have an innate desire to build. To create. To make.

It feels so good to create something from nothing. To make something. To produce. To imagine. To invent.

Whether it's a song, a story, a house, a painting, a bird box, or a picture — **human beings are hard-wired to create**. It's in our DNA.

Don't fight it: **embrace it, accept it, and relish in it.**

And then use it as fuel to create. To make *your* thing.

The feeling of satisfaction after stepping back and looking at your thing is gratifying. Building something that will last is purifying. And inspiring others to build *their* thing is unifying.

Go build your thing. And in the process, inspire others to do the same.

Figure 1. *Humans are hard-wired to create* online work by Devin Arrigo [Arri]

3. WHAT ARE GOOD PATTERNS?

There isn't a definition of pattern quality that is widely accepted in the pattern's community. That is not to say that we do not have agreed guidelines for pattern writing—like choosing a short name reflective of the outcome that can be used in natural language sentences or providing three examples of usage to validate the solution. We also have well-known templates and styles for pattern descriptions. But none of these define or drive the quality of a pattern or a pattern language.

In his book *Software Requirements* [Wie], Karl Wiegers defines desirable characteristics of both individual requirements and requirements collections (Table 1).

Criteria individual requirements	Quality Definition
Complete	Each requirement fully describes the functionality to be delivered
	Each requirement contains all the information necessary for the developer to design and implement that bit of functionality
Correct	Each requirement accurately describes the functionality to be built (decided by user representative)
	Each requirement complies with its parent system requirement
Feasible	Each requirement can be implemented within the known capabilities and limitations of the system and its operating environment (decided by developer)
Necessary	Each requirement documents a capability that the customers really need or one that's required for conformance to an external system requirement or a standard
	Each requirement is traced back to a business need (use case, high level requirement)
Prioritized	Each requirement has a defined priority
Unambiguous	Each requirement is stated in a simple, concise, straightforward language appropriate to the user domain
Verifiable	Each requirement has acceptance criteria defined
	Each requirement has a defined method of verification (test cases, inspection or other)

Criteria requirement collection	Quality Definition
Complete	The set of requirements define a whole where the user workflows within scope can be fully executed
Consistent	The set of requirements do not have conflicts between individual requirements
Modifiable	The set of requirements is managed in an ALM system with version control and change management
Traceable	The set of requirements have an appropriate level of granularity that allow simple linking to business features, tasks, and test

Table 1. Characteristics of Individual Requirements and Collections of Requirements

While Wiegers' characterizations don't exactly resonate with patterns' qualities, they do inspire us to seek out desirable qualities of patterns and pattern languages. What makes a pattern good *is* subjective. But in our estimation, a pattern's goodness is more about that pattern's effect than how it was written.

Good patterns enable you to create something that is not obvious. Good patterns also enhance communication by introducing conceptual solutions that have some degree of complexity that can easily be referred to by name using natural language—they are unambiguous but not simplistic.

A pattern language as originally defined by Christopher Alexander et al in *A Pattern Language* [Alex], is an organized, coherent set of patterns, each of which describes a problem and the core of a solution that can be used in many ways within a specific field of expertise. In that sense, according to Alexander, pattern languages are "complete enough" and self-consistent.

Good patterns offer a high degree of freedom in their application while maintaining their value/intention. While patterns are not precise recipes, they aren't complete or accurate in the sense that Wiegers talks about. Instead, they are constrained.

A pattern language is more than a mere assemblage of individually useful tools; instead, a pattern language attempts to express the deeper wisdom of how to bring aliveness to a particular field of human endeavor, through applying interconnected patterns to create something of quality. According to Alexander, "Every society which is alive and whole will have its own unique and distinct pattern language. Every individual in such a society will have a unique language, shared in part, but which as a totality is unique to the mind of the person who has it. In this sense, in a healthy society there will be as many pattern languages as there are people—even though these languages are shared and similar."[Alex]

To us, the really good patterns are those that become our "constant companions," that is the patterns that we actively apply in our daily work and that inform and influence our actions again and again. More than being memorable, these patterns keep giving us fresh insights and additional knowledge. Each use of them provides new results that evolve and deepen our understanding. In a way, these patterns grow on us and with us year after year.

We use patterns to create—software architectures, product solutions, and our organizations. The quality that we value the most in patterns is the ability to create new things of high value—defined as pattern *generativity*. While we appreciate pattern descriptions that use eloquent language, have clear explanations, good structure,

and nice illustrations, we can live with "good enough" in all of these aspects if the pattern or pattern language serves as a catalyst for making consequential creations.

4. GENERATIVITY AND PATTERNS

For those of us who have been around for a while (or longer) we feel a profound responsibility to ensure the patterns that we provide are good. Generativity has been bandied about in the pattern's community as a desirable property of patterns or pattern languages. Is generativity what makes patterns good? What does it mean for a pattern or pattern language to be generative?

Generativity, as defined in the Cambridge English Dictionary, simply means, "the quality or ability to produce something new."

Applying a pattern always creates something new. Does this imply that all patterns are inherently generative? We don't think so. We believe that pattern generativity means something more.

In *The Nature of Order, The Process of Creating Life* [Alex2002], Christopher Alexander reflects how patterns ideally should be discovered and applied: "Much of our early work implicitly made use of the idea that good patterns were to be derived, somehow, from existing culture, thus ensuring a relation to the subtleties of culture variation, and preserving things that were good and important... Was there, indeed, any way in which one might, by observation of culture as it is, decide in what direction it ought to go, in the future?"

In Alexander's estimation, the goal of good patterns isn't only to create some new thing, but rather to create a better world that respects the existing culture while improving the quality of life.

The Peruvian Experiment, carried out in 1969 by Alexander and three colleagues attempted to do just that. In an intense, reflective process, they observed Peruvian life and culture while designing low-cost homes for Peruvian families. They identified and documented the centers (e.g., the patterns) they used. The criteria for selecting a pattern to be part of their Peruvian pattern language was its relative "strength" and its perceived effects on the culture and lives of the inhabitants. According to Alexander, they "[got] so deeply into the situation that we could feel, in our bodies, just which ones [centers/patterns] needed to be there."

It is telling that Alexander, and his colleagues, didn't solely rely on their gut feelings to judge the quality of their designs. As they wanted to respect Peruvian culture (and not impose their own aesthetics), they sought feedback from potential residents on their emerging designs. The architects found this design process intensely gratifying. Alexander remarked that, "culture-borne centers play a genetic role, not unlike the role played by genes in an organism. They describe what is—in a deep, inner sense. And they also describe how the world can be generated, to become congruent with people's inner feelings, aspirations, habits, and society."

Generative patterns are ambitious. It is insufficient to simply create new things using patterns if we care about bringing more life and wholeness to the world (or more precisely the part of it that we are designing). We should identify and apply impactful patterns that respect the existing culture while improving the quality of life of those who use what we make.

But how do Alexander's views on pattern generativity relate to the patterns that software folks and others in our community write about? Alexander's goals seem rather lofty and out of reach.

We looked to early discussions from members of the software patterns community for clues on aspects important to software pattern generativity.

In his book, *Patterns of Software: Tales from the Software Community* [Gab], Richard Gabriel writes, "Generativity is an interesting trait. Typically, something is generative when it produces the generated quality indirectly." One of his examples is taken from the domain of tennis about how to best hit a tennis ball: "... you should not concentrate on hitting the ball at the point of impact but, instead, hitting a point beyond the ball in the direction the racket is moving. The purpose of this advice is to avoid the effect of the muscles trying to slow down or stop at the point of impact. ... Such advice is generative: The goal is to hit smoothly and with full power, but the goal is not part of the advice. Rather, the advice is to do something else which has the effect of achieving the goal."

This view of generativity resonates with statements in a Portland Pattern Repository wiki discussion [CM], in which Jim Coplien claimed that generative patterns and languages lead to "emergent behavior... that work indirectly; they work on the underlying structure of a problem (which may not be manifest in the problem) rather than attacking the problem directly."

It should be noted that the author of this type of generative pattern will need to have the intended quality goal in mind when writing the pattern. This approach to generativity is very much present in the works of Alexander. Richard Gabriel points to Alexander's pattern, *Alcoves*, as a good example. By creating alcoves in

larger rooms, families can be together even when they are doing different things, with the overarching (indirect) goal being the wellbeing of the family.

In another essay written at that same time, Jim Coplien observed that, "The structures of a pattern are not themselves solutions, but they generate solutions. Patterns that work this way are called generative patterns. A generative pattern is a means of letting the problem resolve itself over time." [Cope]

Gerard Meszaros remarked that: "[a pattern] is more interesting if it covers a range of possible solutions and it leaves the user in control of their own destiny." He also noted that, "Since the user of the pattern is best equipped to understand the context and therefore decide[sic] what forces to optimize at the expense of others, the user is left in a position to determine their own destiny. This interpretation differs from the Alexander's and Coplien's explanations where the emergent behavior is an implicit effect of the pattern. Mezsaros' statement welcomes the creativity of pattern users and opens the solution space for variability.

In *Pattern Hatching: Design Patterns Applied* [Vlis], John Vlissides provides a straightforward definition of pattern generativity while debunking the misconception that "patterns 'generate' whole architectures." He writes: "The generative aspect of patterns gets discussed periodically in the pattern forums. As I understand it, generativity refers to a pattern's ability to create emergent behavior. That's a fancy way of saying the pattern helps the reader solve problems that the pattern doesn't address explicitly."

Patterns with a high degree of generativity work at a fundamentally different level (more broadly or deeply) than patterns with less generativity. Patterns with low generativity merely provide straightforward solutions to directly address specific design problems. Patterns with more generativity, on the other hand, while solving problems also have the potential to change the problem landscape. They create the potential for resolving even more significant problems. They have both a short term and longer-term positive impact.

Less generative patterns tend to be practical and relatively straightforward to apply. They do not require too much from their user in regard to prior knowledge or domain experience.

But people's needs differ. Some want to get to a "good enough" design implemented quickly and are either unwilling or unable to spend hours researching and experimenting to find better designs—those with potentially greater impact and longer term (and often unexpected) benefits. To those hurried designers, their immediate goal is to find reasonable, "proven" solutions that they can reuse without much effort. And often that's good enough. One risk with naively using patterns that aren't particularly generative, however, is that they can lead to mediocre designs that miss the point.

Using generative patterns skillfully requires expertise, experience, experimentation, and reflection. They don't offer quick fixes. Generative patterns demand more of the designer.

Jim Coplien asserted that, "...few published software patterns exhibit generativity." [Cope] Looking back over nearly 30 years of published software design and process patterns that have been written since Coplien made this assessment, we find this still to be the case.

Why is this?

It is rare for any single pattern on its own to be especially generative. For example, the patterns in *Design Patterns: Elements of Reusable Software* [GHJV] provide small ways to structure object-oriented implementations to be more extensible and maintainable. Each pattern is a small building block that solves an immediate design problem. Some patterns provide hooks that allow for future design extension (as long as that extension fits into the patterned structure created for it). Others hide unnecessary details or reduce dependencies between parts of the system. Instead of contorting code to use some other code that doesn't fit the existing design, create an *Adapter* which provides a cleaner interface. Have that *Adapter* call that ugly existing code—hiding details that would compromise the design. Or, hide some existing complexity by introducing a *Façade* which presents a narrower interface.

It is up to the designer to pick and choose among these patterns as they see fit.

While individually, these patterns are useful, these and many other software architecture or design patterns are like stock materials that can be used to solve a specific design problem in a particular way. They shape small bits of software structure. While these are all good patterns, their collective use doesn't automatically add up to something more profound. Applied consistently and repeatedly, and with skill, they can lead to a more coherent, comprehensible design with regular (repeating) structures. But it is a stretch to say that these patterns are especially generative. They have no ambitions for restructuring the nature of the software design problem.

So, rather than considering individual patterns as being particularly generative, we think it is more fruitful to consider the impacts of groups of related patterns or pattern collections, and how it is that they work together towards creating larger structures and improving the quality of a particular aspect of the world. Can specific groups of related software design patterns be considered generative?

John Vlissides' remarks about what makes patterns generative resonate with our experience: "Some of what I've read suggests that true generativity makes this happen almost in spite of one's self. To me, the key to generativity is in the parts of a pattern dedicated to teaching—the forces and their resolution, for example, or the discussion of consequences. These insights are particularly useful as you define and refine an architecture. But patterns don't generate anything—people do, and they do it only if both they and the patterns they use are up to snuff. Moreover, patterns are unlikely to cover every aspect of an architecture. Show me a nontrivial design and I'll show you lots of design issues that no pattern addresses. Perhaps they are not common or recurring issues, or if they are, they have yet to be written up in pattern form. In any event, it's up to you to fill the whitespace between patterns with your own creativity."

The patterns in *Design Patterns: Elements of Reusable Software* claim to provide only some of the elements useful in creating well-structured object-oriented software. They aren't particularly generative. But by skillfully and carefully employing such small structures (classes) along with other design techniques such as those found in *Object Design: Roles, Responsibilities, and Collaborations* [WM], designers can end up with a better object-oriented design. In the hands of skilled designers, those software design patterns can improve both the wholeness of the system and the lives of those who work in the code. But that requires ongoing design attention and curation, and a larger design vision than what patterns provide.

While collections of software design and architecture patterns may not be stunningly generative, they can have positive impacts. And yet, they seem far from meeting Alexander's lofty goal of profoundly changing the world for the better. Should we creators and users of these kinds of patterns feel good about our use of patterns, even as we fall short of Alexander's aspirations?

We think so.

Patterns are "a thing and a process" [Alex79]. As a thing, we want a pattern to have the quality desired (and admired) by its users. As a process, we want a pattern not only to enable the creation of this thing with quality, but to enable the creator to feel pride and fulfillment during the process—a process that is intellectually stimulating, giving the designer freedom to experiment, adapt, and learn. Because creativity and generativity are closely related, we think that patterns and pattern languages can only have a profound impact when their users pay particular attention to generativity.

Our Magic Backlog patterns were written with the intent of improving the flow of and accuracy of information among people working together on a long-lived software product development effort. Structuring a product backlog indirectly impacts the team by improving communication and information accuracy, providing better answers about the current status and quality of the product development efforts. When we initially wrote our patterns, our goal was to provide guidance on how to structure and maintain a backlog enabling it to be a useful tool. We didn't yet fully appreciate how these patterns could also positively impact the team.

As the team and the product's needs evolve, the way the backlog is organized and managed needs to change accordingly. While individual backlog patterns don't offer quick fixes for inaccurate information or poor communication, successively applied, they do have a larger impact than simply the backlog itself. They enable people to work together on a large engineering effort without stepping on each other's toes.

Upon reflection, we believe that our collection of Magic Backlog patterns is generative. And yet, to actually improve product development, these patterns require those who use them to pay ongoing attention to the needs of the people they will impact. They also require experimentation, refitting, and adjustments over time in order to continue to meet the team's needs. Creating a well-structured backlog doesn't happen at once. It takes time, and it takes people who "are up to snuff."

5. THE IMPACT OF CULTURE

Patterns as a simplistic set of instructions—do this, then this, then this—are of limited value, especially for those patterns which deal with human interactions. The problem is that in the hands of less reflective users, these kinds of patterns often lead to mechanical application rather than a team practice of experimentation and evolution into a highly collaborative and well-functioning unit. This is what has happened to too many teams trying to implement agile and lean processes. Agile processes are built on a very clear value system. Their founders spent a lot of energy to make the underlying values and principles the core of their teaching. But many organizations have skipped lightly over these aspects, instead focusing only on the practices. Subsequently, they failed in their agile journey. Those who succeeded took a more thoughtful path—they paid ongoing attention to the values and principles that are integral to the agile culture.

To successfully use any pattern language, we can draw upon a lesson from the agile software community: be explicit about the value system and the principles that form the foundation for the practices that the organization

(and the team) are built on. The same principles apply to patterns: Make sure there is alignment of your values with the value system underlying the pattern language you are using.

Patterns are assumed to work in a given context. But what context is close enough for a pattern to work? Will solutions that work in one organization be readily applied elsewhere? Different organizations may look similar to an outsider—same type of work, same technologies and tooling, similar team size and roles on the team. But the value systems in the organizations may be fundamentally different. As an example, consider the fascination with "The Toyota Way" in the early 2000's—creating Obeya rooms (war rooms) and applying other Toyota practices unsuccessfully in US companies with cultures so different from the Japanese (it can be argued that the Toyota Way caused as many issues as it solved even inside Toyota) [Wiki].

Furthermore, the intricate value system of a team will change over time. It changes with team maturity, and it changes every time team members leave or join. This means that the context is ever-changing. Also, it has facets that are practical and observable, but also facets that are hidden and not well understood. The implications of this shifting context that can only be partially perceived is that patterns that seem like the right solution on the surface can be less than optimal or plain wrong. And vice versa, patterns that seem to be out of scope could actually be helpful.

Although values are embedded in patterns and pattern languages, they aren't often deliberately called out. In hindsight, we realize that we only hinted at the values underlying our backlog when we suggested the target audience for our patterns—teams building complex systems that need to be supported for a long time. Our values were not clearly spelled out as we began our pattern writing because we were in the process of coalescing them as we found and refined our patterns. These values became clarified through our writing: We value accurate and timely information, a retained history of the product development effort, access to relevant product and design documentation, an evident structure to the ongoing product development effort, support for quality assurance and testing, and clearly defined responsibilities for maintaining the backlog that can evolve with the team.

We find that particularly generative patterns are more closely connected to the inner workings of the organizations where they are used. As such, they require more from the users in studying, understanding, and adapting the patterns to their own unique situation. Experience, personal heuristics, and culture play a large role in the application of generative patterns to get the best out of them. If your values significantly differ, then our Magic Backlog patterns may not be for you.

6. MULTIPLE PATTERN LANGUAGES

As we've taken a closer look at our Magic Backlog patterns to understand their generative potential, we think it is also important to recognize better-known people and process related patterns, namely the Organizational Patterns and Fearless Change Patterns.

The original Organizational Patterns were bundled into a book titled *Organizational Patterns of Agile Software Development* to appeal to agile software developers [CH]. Later they were reflected in *A Scrum Book: The Spirit of the Game* [SC]. These authors identified their organizational patterns empirically long before agile or Scrum came on the scene. Nothing about many organization patterns constrain them to any particular software development process (including agile processes) or specific kinds of organizations. Consequently, we find them to be broadly useful for evolving project and product management, software development practices, and the structures of organizations in response to ongoing change.

Fearless Change patterns were written for individuals looking to introduce new ideas into their organizations. Initially published in papers at PLoP, these patterns continue to evolve. They have been collected into two books, *Fearless Change: Patterns for Introducing New Ideas* [RM] and *More Fearless Change: Strategies for Making Your Ideas Happen* [MR]. And now there is a curated website, https://fearlesschangepatterns.com, that offers additional resources and updated information.

We consider Organizational Patterns and Fearless Change Patterns to be good examples of generative patterns. Because the cultures (e.g., the values, principles, behavior, and beliefs) that lie underneath these pattern collections/languages are highly aligned, their individual patterns can operate together in creating solutions that are richer than what could emerge from each individual language.

The Magic Backlog patterns share the same culture, enabling multi-language pattern interactions between all three languages:
1. In a team where the *Developer Controls Process* the team *Involve(s) Everyone* to define the *Backlog Rules* that help them ensure the consistency and correctness of their backlog items that they depend on for their internal team workflows.

2. The new UI Designer on the team realizes that most of the team members are unfamiliar with UX workflows and *Ask(s) for Help* from a seasoned UI Designer from another project and the assigned *Surrogate Customers* on her own project team to create the *People* backlog items that represent the users[1].

Taken together, we find that the organizational and change patterns offer a powerful combination of tools for individual and collective action. We now perceive that organizational structure is more fluid than we initially thought possible and recognize the power of individuals to affect organizational change. Not only have these patterns been effectively used to change organizations we've been part of, but they've also changed us. Some of the patterns have become our "constant companions" as part of our own toolbox for organizational change efforts that we have been involved in. Noë claims that art and philosophy (and for us this includes patterns) have such an important place in our lives because they provide ways to "work ourselves over and make ourselves anew, individually and ensemble."

As we have observed, there are times where patterns from multiple languages work well together, and are needed together for completeness. At the same time we also find gaps in pattern languages that leave the creator in need of adding their own. As John Vlissides points out: "... patterns are unlikely to cover every aspect of an architecture. Show me a nontrivial design and I'll show you lots of design issues that no pattern addresses. Perhaps they are not common or recurring issues, or if they are, they have yet to be written up in pattern form. In any event, it's up to you to fill the whitespace between patterns with your own creativity." [Vlis] It is also up to you to adapt and adjust the solution to fit your exact needs, to derive the essence of any pattern and create your own implementation, and to combine together patterns from various languages to create your own.

In planning her new house, Lise consulted *A Pattern Language* by Alexander [Alex] and found several patterns that are a direct match with the family's expressed requirements, for example *Radiant Heat (230)* and *Sleeping to the East (138).* But she needed to add her own patterns to support elderly or disabled homeowners, for example *Low Storage for Accessibility*. And there were patterns that are incompatible with her pets, like *Indoor Windows (194)* which is dangerous to her parrots that are allowed to fly loose in the house. Now there may be other pattern languages that can help fill the void if they can be identified. But more likely, many of the solutions, although proven over time, are not documented and the spaces must be filled by the combined creativity of the architect and the user.

7. GENERATIVITY AND LEARNING

If your goal is to quickly solve an immediate problem, you only need to learn enough about a pattern in order to use it. But when you are open to re-examining your underlying beliefs and values, which means that you are engaging in Double Loop Learning (Figure 2), you are likely to also want to adjust those values and beliefs [Arg]. Generative patterns would be more effective guides for this kind of reflection and learning if their values were more explicit.
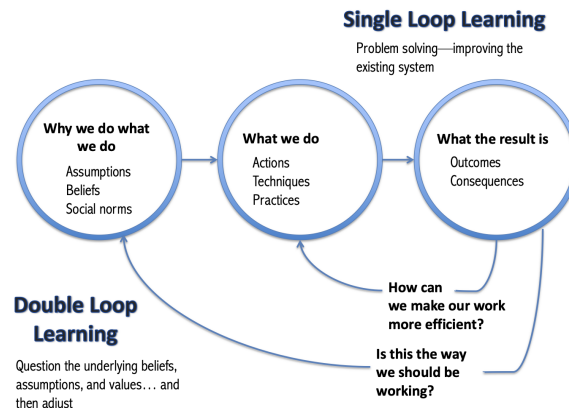

Figure 2. Single and double loop learning.

---

[1] *Developer Controls Process* and *Surrogate Customers* are from the Organizational Patterns
Involve Everyone and *Ask for Help* are from Fearless Change Patterns
Backlog Rules and *People* are from the Magic Backlog Patterns

Single loop learning is focused on solving an immediate problem; double loop learning entails examining beliefs and assumptions and then taking appropriate actions.

In some ways, generative patterns and pattern languages, along with their values, can coach you through a thought process that enables you to discover a solution you believe in while allowing for ongoing learning and reflection.

We believe that some patterns can support the double loop learning process by being generative in nature while at the same time being able to be applied simply. Maybe really good patterns serve both purposes—they provide a basic and direct route for straightforward use while also being valuable to those who are looking for deeper quality. We do not mean to imply that these patterns need to provide both a simple and a more complex solution, but rather that they might serve a dual purpose in speaking to both the novice and to the experienced user who will see more subtleties and sophistication in the pattern descriptions.

This is what Lise calls "SpongeBob patterns." There is a story behind this: Many years ago, her family rented a cabin for Easter vacation. They brought some DVDs for entertainment, including the first SpongeBob movie for the 5-year-old. It should be added that Lise had a very negative opinion about a sponge cartoon and thought it ridiculous with all the backpacks and t-shirts and whatnot with the SpongeBob theme. Really! But being stuck with a single TV in the cabin the family sat down to watch the movie together. To their total surprise they all loved it—the old genetics professor grandfather, Dad the historian who was not much into cartoons whatsoever, the busy mom, and the aloof older brother aged thirteen. They all laughed so much the five-year-old got a bit upset—it was after all his movie! The humor worked on many levels at the same time (Figure 3).

Figure 3. Reddit thread about a Sponge Bob episode [Red]

SpongeBob patterns have this quality of working on several levels. As an example, consider the first pattern in our Magic Backlog Patterns that deals with how to structure the backlog. Here is a short version:

Pattern: *Frame*

How do you organize the main structure of the backlog to best support a variety of users?
You want a backlog that supports the extended development team. You must take into account the different users and support their various activities. Their needs can be quite different. The main backlog structure should represent the overall product in a way that all users are comfortable with. They should be able to easily navigate from a high-level overview to details.

Choose a backlog structure that represents a functional breakdown of your system.
Create a hierarchical structure and link items in this structure in a way that best represents the product to the backlog users. A functional structure is a model that most likely aligns the understanding for most roles on the development team. Also, a functional structure typically takes shape earlier in the development cycle than an architectural view. Any primary structure for your backlog should be able to represent relationships between other backlog objects since any activity should be traceable to a business requirement.
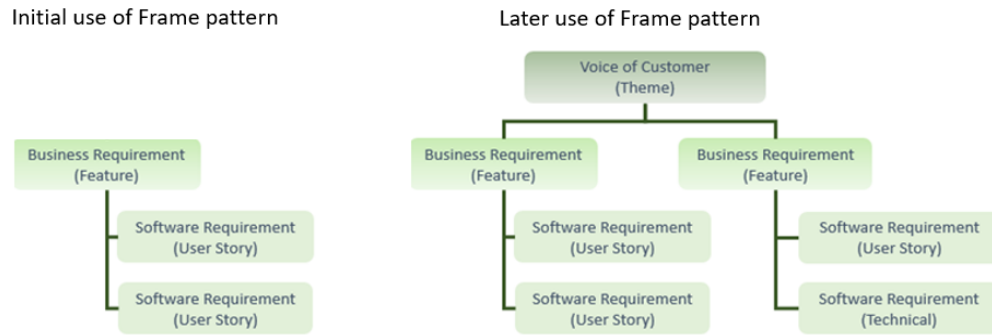
Figure 4. Repeated use of Frame pattern

The first structure of the backlog only has two levels of requirements. Later on, the *Frame* is restructured with more levels (Figure 4). Repeated use of the pattern will add additional backlog item types like risk, test, defect, etc. into a larger structural model. The patterns grow with the maturity of the product and with the expanded workflows of the team. Each time the pattern is used it builds something new that reflects the additional needs and further understanding of the team. Single loop learning is used for immediate improvements of the backlog, while the double loop learning impacts the team's collaboration and internal workflows, which again informs the further development of the backlog structure.

8.   USING GENERATIVE PATTERNS

A pattern language that supports evolution allows for an iterative process where the patterns are gradually applied in their full complexity—unfolding over time as the team needs more sophistication and are ready to benefit from this richness. This implies that the patterns themselves must somewhat work for novices, while mostly being aimed at people with more experience. Our backlog pattern collection does offer proposed sequences that gradually take the creators of a backlog from the basics to more advanced practices. But we should emphasize that these patterns are meant to be employed and refined over time ("Rome was not built in a day"). As seen in the example above, the backlog *Rules* will change as the backlog *Frame* gets more complex. The roles who make the *Rules* may change too, for instance if the initial project grows into a large program.

For a backlog, the evolution of applying, refining, and adapting patterns (and even adding additional ones) could look like this:

In the early days the team is doing fine with the basic application of the patterns. As shown in figure 2, the backlog *Frame* is simple, there is a limited number of backlog item types, and a limited number of *Connections* (relationships) between item types. The *Answers* (insights) that the team depend on are simple and dashboards rather basic.

As the product becomes more complex, the *Frame* may need to be deeper and wider. Additional backlog item types are added to the backlog with new *Connections* defined. With the larger product comes a larger team. In the same way that not every team member is driving the main architecture of a solution, so too is the task of constructing the backlog and ensuring that it serves the needs of the team. Most likely this activity is done by only a few people (maybe the product owner and the software project manager). The people that take ownership of managing the backlog constitute a backlog Community of Practice (CoP), or curatorship, within the team. A backlog CoP is likely interested in utilizing backlog patterns as a way to accelerate their solution and is also possibly interested in the quality of the backlog and in being part of a larger community.

As the development effort matures, solutions are tested and either incorporated or discarded. Solutions will be adapted to the context and to the value system of the organization. The larger and more distributed the product team becomes, the more need there is to document the backlog practices and ensure team members understand and buy into the "rules of engagement." Although there are a smaller number of people who own the backlog and are the keepers of its quality and integrity, it is still important that all who "live" in the backlog and are contributing to its contents are informed users. Just like the people who live in a house do not need to know the architectural patterns that were applied to design it, they still need to understand how to "use" it for it to work best for them. And likewise, the architect needs to understand the people who will live there to design a

house that works best for those who dwell there. The curators become the CoP core and the users become the outer circles of the CoP.

While the crafting of a pattern language should strive towards generativity, the real validation comes in observing the language in use. Only when there is an active community of users—the CoP—will the language be able to adapt and solidify. At this point it has created something more than a direct solution—it has created the necessary support for sustained evolution.

9. DESIGNING GENERATIVE PATTERN LANGUAGES

In our discussions about generative patterns, we also came to realize that a pattern language consists of core patterns that drive the solutions forward, and lesser patterns that have more mundane supporting roles. In the Magic Backlog patterns, the backlog *Rules* is a core pattern that influences not only the quality of the backlog but also the quality of the internal team workflows. It has a strong generative property. A pattern like *Connector* is one that we would classify as supporting; it does not have the same ability to create something more than its basic purpose. This thinking has led us to focus more on the generative qualities of an overall pattern language than its individual patterns.

When you are structuring your knowledge into patterns, you are in effect designing a language. To be clear, this is not to be confused with trying to invent patterns! But you have a body of knowledge, and you need to consider how you can make this as accessible and useful to others as possible. And how it can bring the most value. Christian Kohls remarks that one key to generative patterns is how they communicate: "Rather than telling us exactly what to do step by step (like a micro script) a generative pattern tells us how to *react* to the forces." [Kohl]

Generativity isn't prescriptive. Kohls further observes: "A path is followed and created by performing a sequence of steps; it is a process in which the thing—the path itself—is generated. A particular hike along a path unfolds in the process of walking. It cannot be planned fully in advance. If a stone is on the road, a hiker needs to react, and the particular course is adapted accordingly. If the hiker spots a beautiful flower or butterfly he stops at unpredictable times. A map of paths does not prescribe the exact sequence of steps but rather offers directions and constraints. The hike is volatile; each step is a transformation of the current situation. At any time, the current context needs to be re-evaluated to account [for] the local forces. A path description that is generative tells you how to proceed in the sequence of circumstances: 'when you see the big oak tree, you should keep right until you find a place in the river that is not very deep, so that you can cross it barefoot.'

Instead of having a geographic map with a bird's eye perspective, demotic instructions or sketches on a napkin concentrate on the actions and milestones that are important to follow the path successfully. In that respect informal descriptions are quite precise because they take into account what actually matters in the given context. The form of the description—watching out for landmarks, describing conditions for the next operation—provides context-based instructions."

When Lise, at her work, started being serious about the backlog (when they had grown from a small tribe working on a single product, to a large program supporting multiple products and having teams distributed across the globe), she tried to find guidance on good quality backlogs. But neither internally, nor externally did she find help above the level of backlog basics provided by the tool vendors (for example, the ADO documentation from Microsoft).

But if she had found something like our backlog patterns, would she and her colleagues have been able to get the best out of them? Such a pattern collection would have been helpful, but we are not sure she and her colleagues would have been able to take the patterns and accelerate the iterative process of improving not only the backlog but also the team processes. Most likely they would have had a mechanistic implementation similar to the poor Scrum implementations that give agile development a bad reputation.

So then, what would the design process look like for the gradual development of a pattern language with the generative qualities we are looking for? We think it starts small and practical, and then gradually matures into more sophistication. Practices start being recognized and repeated, and eventually they are also documented as patterns. The pattern names over time become part of the vocabulary, and a language starts taking shape. For the language to take on a more significant role and become known throughout an organization, or in the software community at large, it needs a Community of Practice (COP) composed of people who are interested and enthusiastic about using, refining, and driving the language forward. This process is shown in Figure 5.
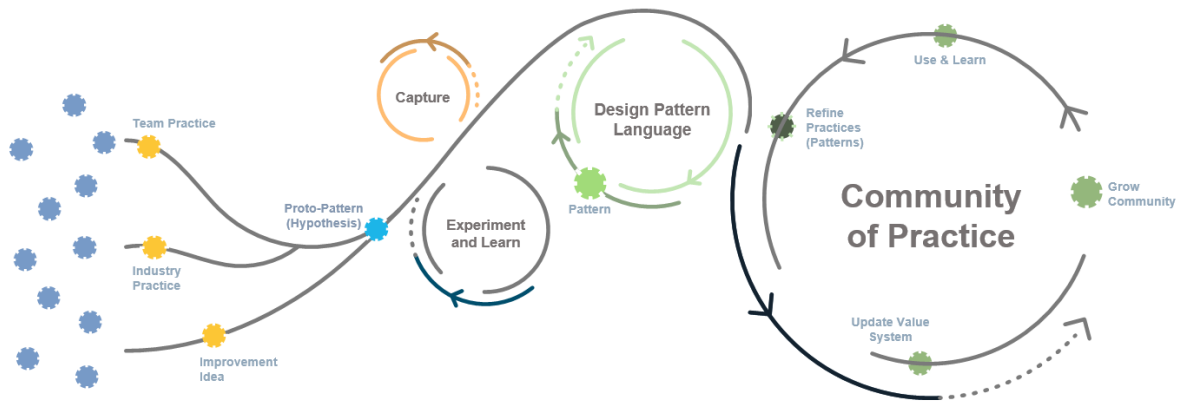
Figure 5. The growth of a pattern language

Patterns and pattern sequences are used and reused over and over by the same people, and by new users. Learning is incorporated back into the language where some practices evolve and strengthen while others may fall out of use and be removed. Just like the architecting of a software system, the pattern language must be restructured, and patterns revised and reworked to stay current and useful.

As a pattern language grows and matures it becomes easier to see how it complements and interacts with other pattern languages in the same or related domains. In Figure 6, we try to visualize this with pattern domains (Organization, Process, Quality, and Design) with pattern languages within the domains. Not only do the domains and the patterns create a larger body of knowledge together, but there is a whole ecosystem of understanding that incorporates many sources of knowledge, some that are thoroughly documented and others that are more fluid and based on human interaction.
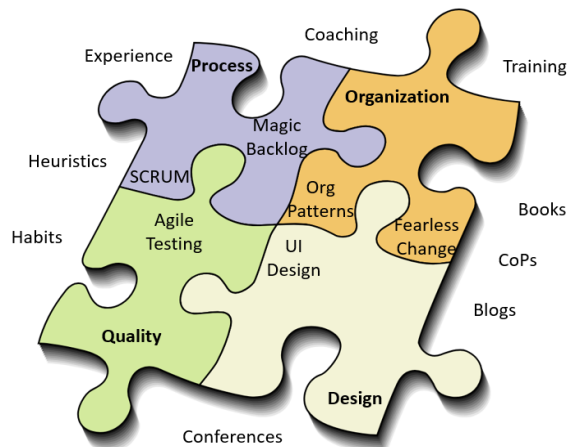

Figure 6. Illustration of Pattern Languages in an Ecosystem of Knowledge

Most pattern authors will focus primarily on their own pattern language early on in their work, when the bulk of the patterns are being captured and matured. Gradually they will become aware of relationships between their patterns and other languages, which enables them to strengthen and improve their pattern language by pulling in and referring to additional sources.

From the early discussions on generativity in the software community we see two schools of thought that a patterns author should be consciously aware of in their pattern writing. First, there is the idea that a generative pattern will create emergent behavior that is the actual goal of the pattern, but that is never explicitly stated in the pattern solution. Unless this implicit goal is somehow present in the forces or resulting context descriptions, this kind of generative pattern leaves the user uninformed about the deeper reasons behind the pattern; this knowledge stays with the pattern author. One can argue that this puts less demand on the user, but also that it

doesn't encourage the user to actively explore solutions and consciously grow their expertise and understanding.

Alternatively, the pattern author can emphasize the ability of a generative pattern to teach, while providing a high degree of freedom in the solution. This may make the pattern more difficult to apply, but also way more interesting to a knowledgeable and experienced user.

We are not saying that one way of achieving generativity is better than another. It depends on the characteristics of the intended pattern users and the goals of the pattern author (for example, whether to teach or to offer simple, practical advice).

And yet, regardless of the pattern author's intent, users of patterns or a pattern language will come at the material from different perspectives. They'll be trying to tie in these patterns to what they already know and do, and testing and adapting the solutions to their needs. It is only through this iterative, messy, unpredictable process that the patterns get refined, and that the generativity really comes into play in creating new solutions.

10. CONCLUSIONS

Generative patterns tend to resolve design problems while supporting a wide range of possible solutions. Sure, they resolve problems, but they also have the potential to radically change the problem landscape.

Rarely, do individual patterns have significant impact. Rarely, can any individual pattern be said to result in design breakthroughs. It is only through repeatedly adapting and applying multiple patterns (as found in pattern languages and through experience), paying attention to their cumulative effects, and tweaking and re-tweaking your design that you break out of the ordinary.

Our thoughts on generativity resonate with Parker Richards, staff editor of the Time Opinion who writes, "Inefficiency is generative; inefficiency is where we source our ideas, our inspirations, our conceptions of a world of endless paths and journeys rather than one of monotonous drudgery occupied only in repetitive labor. A life whimsically lived, a society whimsically (dis)ordered, is one that promotes freedom of thought, even as it knows many of the freely found thoughts won't be all that useful." [Rich]

It is as if a force field for positive change has been set in place by generative patterns. Only then can unforeseen design potentials appear. But it is up to you, the maker, to breathe life into and sustain the quality of your design. Generativity isn't a fast burn. And it doesn't follow a predictable path.

So, what exactly is pattern generativity? We have avoided formulating a precise definition because we want to leave room for you, the reader, to think deeply about generativity and join us in conversation. How does pattern generativity help you in creating things of quality?

11. ACKNOWLEDGEMENTS

12. APPENDIX A: SUMMARY OF MAGIC BACKLOG PATTERNS

Pattern name	Description
Frame	How do you organize the main structure of the backlog to best provide the benefits of a quality backlog to a variety of users? Choose a backlog structure that represents a functional breakdown of your system. Create a hierarchical structure and link items in this structure in a way that best represents the product to the backlog users. A functional structure is a model that most likely aligns the understanding for most roles on the development team.
Views	How can the backlog provide representations of a product that is intuitive to a variety of user roles? Create additional backlog structures to reflect alternate *views* of the product, for instance an architectural view and a quality view. Lower-level backlog items can be linked both to items in the functional product structure (the *Frame*) and to items in the alternate structures. As an example, a User Story can be linked both to a main Feature (in the *Frame*) and to a Subsystem (in the architectural view).
People	How can you represent the various aspects of your system's users in a backlog? Create backlog items for personas to cover the dimensions of user profiles and associate the personas with the appropriate functional backlog items. Their descriptions are then readily available for any team member with access to the backlog. Either tag a user story with the name of the persona or link the persona backlog item to the functional item.
Tales	How can you improve the understanding of how users interact with the system and the impact on dependencies between individual user stories? Include narratives that give a free-form representation of product usage in your backlog. Most likely your narrative will span multiple user stories, and the natural level to link it in is to the feature level. The actual text for the narrative is captured in a document which is then uploaded as an attachment to the narrative backlog item.
Usage Models	How can you improve the understanding of how individual user stories contribute to a business transaction or user goal? Enrich your backlog with models that provide a structured representation of product usage. Each *usage model* represents a business transaction or a use of the system as a whole to accomplish a complex task. The purpose of the model is to improve your understanding of how the system is used and provide a tool to prioritize, plan, and verify your product deliveries. Possible models are Use Cases and Business Process Models.
Placeholders	How can you represent partly unknown functionality in your backlog? Create temporary backlog items as *placeholders* to be exchanged for detailed items later, when they have been elaborated. When the detailed items are created, you will want to replace your placeholder backlog item with the new detailed items. If you instead keep the placeholder item and link these details to it, you will increase the levels in your backlog thereby making querying and backlog maintenance that much harder.
Plans	How are the backlog items associated with your plans for delivery? Associate the detailed requirements slotted for the next delivery to an entity representing this delivery. Tools normally associate backlog items with iterations and releases by using a planning-related attribute on backlog items. Backlog contents can then be filtered based on the values of this attribute to produce lists of items for a specific release.

Pattern name	Description
Connections	How can you explore the diverse contents of your Application Lifecycle Management (ALM) system? Create *connections* from other item types to the appropriate requirements backlog items. You want to establish these connections systematically following a defined model, normally linking tests to requirements, defects to both requirements and to the tests that detect and/or verify the defect resolution and change sets to the requirements they implement or defects that they resolve.
Answers	How can your team gain insights about the product from the backlog? Create shared queries and reports that can be reused by your team. The primary focus when extracting information from the backlog should be on the direct development team needs, and not stakeholders. The goal is for the core team to always know where they are and be able to prioritize their efforts on the most pressing work.
Pipeline	How can you ensure that you always have some backlog items with sufficient maturity to enter the development process? Design a process that creates a steady stream of prepared backlog items. The process works as a *pipeline* that steadily refills the backlog with items with enough detail to be meaningful to the developers.
Funnel	How and when do you introduce new product ideas into your backlog? Keep a list of future product ideas to explore that is separate from your Product Backlog. When an idea has been accepted into the product scope and has matured enough to be represented by epics level items, then introduce these into your Backlog. Expect that a good portion of product ideas will never be fully developed. Some may be discarded early after limited investigation either because they cannot be supported by a business case, because they are too costly to develop, or because they just do not fit into the portfolio.
Maintenance	How do you keep your backlog as a reasonably accurate representation of the planned and implemented product? Regularly and consistently maintain the backlog contents. Maintaining the backlog is more than adding details and updating statuses. New contents need to be added as new requirements are elicited. Business priority changes will adjust the user story sequence/iteration planning. A maturing understanding of the product may require refactoring of the structure for the *Frame* and the alternate *Views*. Objects and attributes that the team uses for its planning and metrics need to be updated as the items go through the *Funnel* and the *Pipeline* and then through implementation/verification, making sure that structure and attribute changes caused by new material is consistently applied across the full set of contents.
Shared Definitions	How do you ensure that key information in the backlog that is used to drive internal team processes and to communicate with stakeholders is correct enough to be meaningful? Develop and share a core set of definitions across the project or program so that the attribute values of your backlog items are consistent and can be used for decision making and reporting purposes. Keep these definitions in a shared space that is easily accessible by all team members, like a project or program wiki. But even more important, make sure that these definitions are actively used because the team members contributed to defining them, agree with their definition, and know where to find them.

Pattern name	Description
Rules	How do you protect the backlog from changes that risk adversely affecting key team processes? Create role-based rules for backlog changes and only permit specific roles to make those changes that impact the overall team. Restrictions defined by these *rules* should be only for those backlog items that impact the ability of the project and/or program workflows to run efficiently, and to items and attributes that are part of the commitment to stakeholders. Individual team members should have full control of all other items and their attributes that relate to their own work.
Remodel	How do you deal with a mature product backlog whose structure is no longer efficiently supporting the development team? Remodel the backlog to better represent the new understanding of the product while keeping core backlog items largely unchanged. Changing the backlog *Frame* through modifying the way items are linked to each other still fully preserves the definition of each backlog item. So, an update to better represent the system functionality is most likely an exercise in creating new/updated top-level items while keeping the contents of user stories untouched and linking them to this new structure.
Pragmatic Program Backlogs	How do you manage a program-level view of the work of individual projects with highly different backlog implementations, so that you can better coordinate their work? Construct an additional backlog that has only the *Frame* representation of the product solution to be built. The user stories in this program-level backlog are normally on the epic level and will typically be implemented by several project level user stories. These user stories will belong to multiple projects. Instead of tracking individual product backlog items, this program-level backlog enables you to manage the work at a higher level, focusing on major features and sets of related features.
Linked Program Backlogs	How do you organize and manage the program-level backlog for a program consisting of a set of closely aligned projects when each project has its own mature and extensive backlog and these backlogs are structured differently? Create a program-level backlog where you keep individual project backlogs, and link project backlog items to the program level backlog for traceability. This approach lets the individual projects have their individual backlogs structured to support their own way of working while still being able to automate Answers on product completeness and outstanding issues. The typical backlog items to link would be user stories and test cases in the individual backlogs to the respective features and test suites/test plans in the product level backlog.
Unified Program Backlog	How do you organize and manage the backlog for a program consisting of closely aligned projects when the projects have no current backlogs or backlogs that are small and structurally similar? Define a single, unified backlog shared by all the projects within the program, but allow for projects to apply attributes, tags, and filters that provide them a specialized project level view of the contents. Typically, the program backlog structure has a set of goals, features, and user stories representing the user functionality. Goals and features are most likely shared between all teams, and there is no need to filter them at the project level. The items that you will want to view and work with specifically on a project level are the user stories and the defects.

REFERENCES

[Alex] Alexander, C., Ishikawa, S., Silverstein, M., Jacobson, M., Fiksdahl-King, I., Angel, S. 1977. *A Pattern Language: Towns, Buildings, Construction.* Oxford University Press.

[Alex1979] Alexander, C. 1979. *The Timeless Way of Building,* Oxford University Press.

[Alex2002] Alexander, C. 2002. The Process of Creating Life: Nature of Order, Book 2: An Essay on the Art of Building and the Nature of the Universe. Center for Environmental Structure.

[Arg] Argyris, C. Teaching smart people how to learn. 1991. Harvard Business Review. 69 (3): 99–109.

[Arri] Arrigo, D. (2020, December 21). Humans are hard-wired to create. Retrieved on February 9, 2024 from https://medium.com/extraordinary-humans/humans-are-hard-wired-to-create-ac5894cc604

[CH] Coplien, J. and Harrison, N. 2004. *Organizational Patterns of Agile Software Development.* Prentice Hall.

[Cope] Coplien, J. 1996. *Software Patterns.* SIGS Publications.

[Gab] Gabriel, R.P. 1996. *Patterns of Software: Tales from the Software Community.* Oxford University Press.

[GHJV] Gamma, E., Helm, R., Johnson, R., Vlissides, J. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley.

[HW2015] Hvatum, L. and Wirfs-Brock, R. 2015. Patterns to Build the Magic Backlog. 20th European Conference on Pattern Languages of Programming (EuroPLoP), EuroPLoP 2015, July 8-12 2015, 36 pages.

[HW2017] Hvatum, L. and Wirfs-Brock, R. 2017. Pattern Stories and Sequences for the Backlog: Expanding the Magic Backlog Patterns. 24th Conference on Pattern Languages of Programming (PLoP). PLoP 2017, October 23-25 2017, 26 pages.

[HW2018] Hvatum, L. and Wirfs-Brock, R. 2018. Program Backlog Patterns: Applying the Magic Backlog Patterns. 23rd European Conference on Pattern Languages of Programming (EuroPLoP). EuroPLoP 2018, July 4-8 2018, 22 pages.

[Jul] JuliusOrange437 (2019, September 16). Comments on, Only the OG's will remember. Reddit. Retrieved February 9, 2024 from https://www.reddit.com/r/memes/comments/gnmxja/only_the_ogs_will_remember/

[Kohl] Kohls, Christian. 2013. The Theories of Design Patterns and their Practical Implications exemplified for E-Learning Patterns. [Doctoral dissertation, der Katholischen Universität Eichstätt-Ingolstadt].

[Noë] Noë, A. 2023. *The Entanglement: How Art and Philosophy Make Us What We Are.* Princeton University Press.

[MR] Manns, M. and Rising, L. 2015. More Fearless Change: Strategies for Making Your Ideas Happen. Addison-Wesley.

[RM] Rising, L. and Manns, M. *Fearless Change: Patterns for Introducing New Ideas.* Addison-Wesley, 2004.

[Rich]Richards, P. (2023, October 5). *The New York Times.* Down With Efficiency! (When We Get Around to It.) https://www.nytimes.com/2023/10/05/opinion/efficiency-optimization-whimsy.html.

[SC] Sutherland, J., Coplien, J. et al. 2019. *A Scrum Book: The Spirit of the Game.* Pragmatic Bookshelf.

[Vlis] Vlissides, J. 1998. *Pattern Hatching: Design Patterns Applied.* Addison-Wesley Professional.

[Wie] Wiegers, K. and Beatty, J. 2013, *Software Requirements Third Edition.* Microsoft Press.

[WH2016] Wirfs-Brock, R. and Hvatum, L. 2016. More Patterns for the Magic Backlog. 23rd Conference on Pattern Languages of Programming (PLoP). PLoP 2016, October 24-26, 2016, 18 pages.

[WH2018] Wirfs-Brock, R. and Hvatum, L. 2018 Even More Patterns for the Magic Backlog. 25th Conference on Pattern Languages of Programming (PLoP). Plop 2018, October 24-26, 2018, 17 pages.

[WH2019] Wirfs-Brock, R. and Hvatum, L. 2019. Who Will Read My Patterns? On Designing a Patterns Book for Target Readers. 26th Conference on Pattern Languages of Programs (PLoP). PLoP 2019, October 7-10, 2019, 22 pages.

[Wiki] The Toyota Way. 2023. Retrieved February 9, 2024 from https://en.wikipedia.org/wiki/The_Toyota_Way.

[WM] Wirfs-Brock, R. and McKean. 2002. A. *Object Design: Roles, Responsibilities, and Collaborations.* Addison-Wesley Professional.