Conceptual Engineering in Software Systems

0. Abstract

Taking the *engineering* part of *conceptual engineering* seriously means learning from the methods of other engineering disciplines. Since the concepts that philosophers want to engineer are generally parts of large, complex, existing systems, they are attempting to engage in *systems engineering*. The branch of systems engineering that deals most in concepts is *software systems engineering*. This is a relatively new discipline, having emerged out of less systematic software development practices by borrowing certain key methods from recent developments in architecture. This talk traces the close analogy between architects' *design patterns* and philosophers' *concepts*, then follows the lessons learned in software systems engineering about what makes a good design pattern, how to engineer a better one, and how to evangelize better design patterns after their invention. Two key lessons are a sweet spot in the scope of conceptual engineering projects and the importance of structures of collaboration.

1. Keeping Control: Conceptual Engineering as/in Engineering

1.1 Conceptual Engineering as Systems Engineering

"Systems engineering is an interdisciplinary field of engineering and engineering management that focuses on how to design and manage complex systems over their life cycles…The individual outcome of such efforts, an engineered system, can be defined as a combination of components that work in synergy to collectively perform a useful function. Issues such as requirements engineering, reliability, logistics, coordination of different teams, testing and evaluation, maintainability and many other disciplines necessary for successful system development, design, implementation, and ultimate decommission become more difficult when dealing with large or complex projects. Systems engineering deals with work-processes, optimization methods, and risk management tools in such projects. It overlaps technical and human-centered disciplines such as…software engineering…"[1]

"There are three design metaheuristics.

1. **Inverse model is tractable**: If there is a tractable 'inverse model' of the system, then there is a way of working out in advance a sequence of variations that brings about a desired set of objective values. Here a design is a plan, like a blueprint, and the inverse model is this: once the thing to be designed is imagined, there is a way to work out the blueprint or a plan for how to build the thing. This is the essential up-front design situation, which works best when designing something that has been built at least once before, and typically dozens or thousands of times.
2. **Inverse model is not tractable, but forward model is**: In this case, we can predict the influence of variations upon the objective values, but the system is not tractably invertible so we cannot derive in advance a sequence of variations to bring about a desired set of objective values. This implies an iterative approach, where variations carefully selected according to the forward model are applied in sequence.

[1] "Systems Engineering."

3. **Neither forward nor inverse models are tractable**: There is neither a way of discerning which variations will give improvements in the objective values, nor a way of predicting what will be the effects of variations upon the objective values. Without evolution all is lost."[2]

1.2 Conceptual Engineering in Software Engineering

"Things are built from parts…Software is built from concepts."[3]

"Conceptual integrity is the most important consideration in system design."[4]

Criteria for Concepts: Simple, Coherent, Complete, Usable[5]

Fighting Over Concepts: Programming with Nouns or Verbs?[6]

"[This is] a book of **design patterns** that describes simple and elegant solutions to specific problems in object-oriented software design. Design patterns capture solutions that have developed and evolved over time. Hence they aren't the designs people tend to generate initially. They reflect untold redesign and recoding as developers have struggled for greater reuse and flexibility in their software. Design patterns capture these solutions in a succinct and easily applied form. The design patterns require neither unusual language features nor amazing programming tricks with which to astound your friends and managers. All can be implemented in standard object-oriented languages, though they might take a little more work than *ad hoc* solutions. But the extra effort invariably pays dividends in increased flexibility and reusability. Once you understand the design patterns and have had an "Aha!" (and not just a "Huh?") experience with them, you won't ever think about object-oriented design in the same way. You'll have insights that can make your own designs more flexible, modular, reusable, and understandable—which is why you're interested in object-oriented technology in the first place, right?"[7]

"Designing object-oriented software is hard, and designing *reusable* object-oriented software is even harder. You must find pertinent objects, factor them into classes at the right granularity, define class interfaces and inheritance hierarchies, and establish key relationships among them. Your design should be specific to the problem at hand but also general enough to address future problems and requirements. You also want to avoid redesign, or at least minimize it. Experienced object-oriented designers will tell you that a reusable and flexible design is difficult if not impossible to get 'right' the first time. Before a design is finished, they usually try to reuse it several times, modifying it each time."[8]

2. A Jump to the Left: Architecture from Art to Systems Engineering of Concepts

2.1 Architectural Concepts

"The elements of this language are entities called patterns. Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it in

[2] Gabriel, "Foreword," xv–xvii, quoting Thompson, "Notes on Design Through Artificial Evolution," 19.
[3] Jackson, "Conceptual Design."
[4] Brooks, *The Mythical Man-Month*.
[5] Jackson, "Conceptual Design."
[6] Yegge, "Stevey's Blog Rants."
[7] Gamma et al., *Design Patterns*, 7.
[8] Ibid., 10.

the same way twice. For convenience and clarity, each pattern has the same format. First, there is a picture, which shows an archetypal example of that pattern. Second, after the picture, each pattern has an introductory paragraph, which sets the context for the pattern, by explaining how it helps to complete certain larger patterns. Then…[a] headline gives the essence of the problem in one or two sentences. After the headline comes the body…[which] describes the empirical background of the pattern, the evidence for its validity, the range of different ways the pattern can be manifested in a building, and so on. Then…is the solution—the heart of the pattern—which describes the field of physical and social relationships which are required to solve the stated problem, in the stated context. This solution is always stated in the form of an instruction—so that you know exactly what you need to do, to build the pattern. Then, after the solution, there is a diagram…and finally…there is a paragraph which ties the pattern to all of those smaller patterns in the language, which are needed to complete this pattern...There are two essential purposes behind this format. First, to present each pattern connected to other patterns, so that you grasp the collection of all 253 patterns as a whole, as a language, within which you can create an infinite variety of combinations. Second, to present the problem and solution of each pattern in such a way that you can judge it for yourself, and modify it, without losing the essence that is central to it."[9]

2.2 Architectural Conceptual Engineering

"*A Pattern Language* is the second in a series of books which describe an entirely new attitude to architecture and planning. The books are intended to provide a complete working alternative to our present ideas about architecture, building, and planning—an alternative which will, we hope, gradually replace current ideas and practices."[10]

"This book provides a language, for building and planning; the other book provides the theory and instructions for the use of the language. This book describes the detailed patterns for towns and neighborhoods, houses, gardens, and rooms. The other book explains the discipline which makes it possible to use these patterns to create a building or a town. This book is the sourcebook of the timeless way; the other is its practice and origin."[11]

"*The Timeless Way of Building* describes the fundamental nature of the task of making towns and buildings. It is shown there, that towns and buildings will not be able to become alive, unless they are made by all the people in society, and unless these people share a common pattern language, within which to make these buildings, and unless this common pattern language is alive itself. In this book, we present one possible pattern language, of the kind called for in *The Timeless Way*."[12]

"In the patterns marked with two asterisks, we believe that we have succeeded in stating a true invariant: in short, that the solution we have stated summarizes a *property* common to *all possible ways* of solving the stated problem. In these two-asterisk cases we believe, in short, that it is not possible to solve the stated problem properly, without shaping the environment in one way or another according to the pattern that we have given—and that, in these cases, the pattern describes a deep and inescapable property of a well-formed environment."[13]

"In the patterns marked with one asterisk, we believe that we have made some progress towards identifying such an invariant: but that with careful work it will certainly be possible to improve on the solution. In these cases, we believe it would be wise for you to treat the pattern with a certain

[9] Alexander, Ishikawa, and Silverstein, *A Pattern Language*, x–xi.
[10] Ibid., overleaf.
[11] Ibid., ix.
[12] Ibid., ix–x.
[13] Ibid., xiv.

amount of disrespect—and that you seek out variants of the solution which we have given since there are almost certainly possible ranges of solutions which are not covered by what we have written. Finally, in the patterns without an asterisk, we are certain that we have *not* succeeded in defining a true invariant—that, on the contrary, there are certainly ways of solving the problem different from the one we have given. In these cases we have still stated a solution, in order to be concrete—to provide the reader with at least one way of solving the problem—but the task of finding the true invariant, the true property which lies at the heart of all possible solutions to this problem, remains undone."[14]

"You see then that the patterns are very much alive and evolving. In fact, if you like, each pattern may be looked upon as a hypothesis like one of the hypotheses of science. In this sense, each pattern represents our current best guess as to what arrangement of the physical environment will work to solve the problem presented. The empirical questions center on the problem—does it occur and is it felt in the way we have described it?—and the solution—does the arrangement we propose in fact resolve the problem? And the asterisks represent our degree of faith in these hypotheses. But of course, no matter what the asterisks say, the patterns are still hypotheses…all free to evolve under the impact of new experience and observation."[15]

"*The Timeless Way of Building* says that every society which is alive and whole, will have its own unique and distinct pattern language; and further, that 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…The fact is, that we have written this book as a first step in the society-wide process by which people will gradually become conscious of their own pattern languages, and work to improve them. We believe…that the languages which people have today are so brutal, and so fragmented, that most people no longer have any language to speak of at all…We have spent years trying to formulate this language, in the hope that when a person uses it, he will be so impressed by its power, and so joyful in its use, that he will understand again, that it means to have a living language of this kind."[16]

"Many of the patterns here are archetypal—so deep, so deeply rooted in the nature of things, that it seems likely that they will be a part of human nature, and human action, as much in five hundred years, as they are today. We doubt very much whether anyone could construct a valid pattern language, in his own mind, which did not include the pattern ARCADES for example, or the pattern ALCOVES."[17]

3. A Step to the Right: Lessons from Fifteen Years of Conceptual Engineering in Software Systems

3.1 Characteristics of Good Concepts

"In almost all cases in which the term 'pattern' is used in the context of software design, there is an underlying assumption that we are talking about 'good' patterns—that is, patterns that are complete, balanced, and clearly described, and that help to establish some kind of wholeness and quality in a design. This assumption is so common that the word 'good' is almost always dropped: the word 'pattern' effectively subsumes it. If a pattern, however, represents a commonly recurring

[14] Ibid., xiv–xv.
[15] Ibid., xv.
[16] Ibid., xv–xvi.
[17] Ibid., xvii.

design story— one that varies a little with each retelling—there are not just good stories to tell. Some recurring practices are incomplete, unbalanced, and fail to establish a recognizable wholeness and quality. What they lack in goodness they often make up for in popularity—in spite of usage to the contrary, 'common practice' and 'best practice' are not synonyms. Some patterns have recurrence but not quality. These are 'bad patterns,' a phrasing used by Christopher Alexander in his original framing of patterns and the motivation for identifying 'good patterns.' 'Bad patterns' are dysfunctional, often twisting designs out of shape, leading to a cycle of 'solving the solution,' in which the ingenuity of developers is consumed in fixing the fixes (to the fixes to the fixes…)."[18]

"In identifying and separating good patterns from dysfunctional patterns or nonpatterns, we first need to distinguish between solution ideas that recur and those that are singular, and so tightly bound to their application that whatever their merits as specific designs, they cannot be considered general patterns. We could say that each recurring solution theme qualifies as a 'candidate pattern.' We also need more than just the idea of recurrence in a pattern to make it a useful part of our development vocabulary. We need to know that its application is beneficial rather than neutral or malignant—a pattern with no effect or ill effect. Before hastily branding 'dysfunctional' any candidate pattern that fails to contribute successfully to a design, however, we must also be able to distinguish between patterns that are truly dysfunctional and patterns that have simply been misapplied—in other words, dysfunctional applications of otherwise sound patterns. To assess these qualities reasonably we need to render our candidate pattern in some form that makes them visible. A pattern description must be clear and must make it possible to see the qualities of the pattern—a poor pattern description can obscure a good pattern, selling it short, whereas a good 'spin' can mask a poor pattern, overselling it."[19]

"How can we ensure the context's completeness? An overly general context that acts as an umbrella for many possible problem situations may be too vague, leading to inappropriate applications of a pattern. On one hand, too much detail will probably yield an unreadable pattern prefixed by an exhaustive 'laundry list' of specific context statements. On the other, an overly restrictive context may prevent developers from applying a pattern in other situations in which it is also applicable. They may take the pattern only as written and fail to grasp the opportunity for generalization it implies. One way of coping with this problem is to start with a context that describes the known situations in which a pattern is useful, and to update this context whenever a new appropriate situation is found…It is better to support the appropriate application of a pattern in a few specific situations than to address the possible application of a pattern everywhere—a jack of all trades, but a master of none."[20]

"If, on the other hand, such requirements [on effective solutions] were added to the pattern's problem statement, they would become explicit, and a concrete solution for the problem could deal with them appropriately. The same argument holds for any desired property that the solution should provide. Similarly, there may be some constraints that limit the solution space for the problem, or just some things worth taking into account when resolving it. These requirements, desired properties, constraints, and other facts fundamentally shape every given problem's concrete solution. The pattern community calls these influencing factors *forces*, a term taken from Christopher Alexander's early pattern work. Forces tell us why the problem that a pattern is addressing is actually a problem, why it is subtle or hard, and why it requires an 'intelligent'—perhaps even counter-intuitive—solution. Forces are also the key to understanding why the

[18] Buschmann, Henney, and Schmidt, *On Patterns and Pattern Languages*, 31.
[19] Ibid., 31–32.
[20] Ibid., 44–45.

problem's solution is as it is, as opposed to something else. Finally, forces help prevent misapplications of a pattern in situations for which it is not practical."[21]

"Put simply, the context of a pattern is where it can be applied. But what does that mean? Given that a pattern addresses a problem arising from particular forces, those forces must be connected to the context. If the context is not part of the pattern, then neither are the forces. However, if the forces are part the pattern—which they must be, otherwise patterns just become solutions—they must be rooted in the context and have arisen from there. The context cannot therefore be divorced from the pattern. Yet the context for a pattern considered as a stand-alone pattern is in some way more generic than the context for a pattern considered within a pattern sequence."[22]

"Unfortunately it might not always be possible to resolve all forces completely in a solution: forces are likely to contradict and conflict with one another. A particular force may address an aspect that can only be resolved at the expense of the solution's quality in respect to aspects addressed by other forces."[23]

"the concept in question must be a designed artifact, as opposed to something occurring in nature,"[24]

Properties of [good] patterns:[25]

- Document existing best practices
- Identify and specify abstractions
- Provide a common vocabulary and shared understanding

Pitfalls of [proto-] patterns:[26]

- Over-specificity
- Over-fixity
- Misunderstood context
- Inconsistency

Dysfunctional patterns:[27]

- Rely on forces which are merely the rationalizations of inexpert decision-makers
- Don't provide a resolution of forces

"Just as most useful software evolves over time as it matures, many useful pattern descriptions evolve over time as they mature. This maturity results primarily from the deeper experience gained when applying patterns in new and interesting ways."[28]

"The following three criteria should be followed to ensure that pattern languages support the creation of 'best of breed' solutions and effective development practices:

[21] Ibid., 37.
[22] Ibid., 203–4.
[23] Ibid., 39.
[24] Ibid., 52.
[25] Ibid., 8–12.
[26] Ibid., 13–20.
[27] Ibid., 40.
[28] Ibid., 59.

1. *Sufficient coverage*. Pattern languages must include the right patterns to address the various aspects and subproblems of their subject, otherwise they cannot support the creation of useful software.
2. *Sustainable progression*. Pattern languages must connect their patterns appropriately to ensure that challenges are addressed in the right order, which is essential to creating sustainable designs incrementally and via stable intermediate steps.
3. *Tight integration*. Pattern languages must integrate their constituent patterns tightly, based on the roles each pattern introduces and the inter-relationships between them."[29]

"the notion of *pattern complements* [has] two aspects:

- *Complementarity with respect to competition*. One pattern may complement another because it provides an alternative solution to the same or a similar problem, and thus is complementary in terms of the design decision that can be taken.
- *Complementarity with respect to structural completeness*. One pattern may complement another because it completes a design, acting as a natural pairing to the other in a given design."[30]

"Pattern languages are no exception to the following timeless rule: each pattern language can only reasonably document the experience of the past and the present, and can cover competently only those problem and solution areas of which their authors are aware, or on which they have focused intentionally. Similarly, the quality of a pattern language is only as good as the quality and maturity of its constituent patterns (vocabulary) and the network (grammar) that they form. It is the practical experience gained over time—paired with hindsight—that ultimately confirms whether a pattern language is functional or dysfunctional, what deficiencies and gaps it has, and by what measures these drawbacks and holes can be addressed."[31]

3.2 Engineering Better Concepts

"Treating pattern languages as works in progress demands even more 'gardening' effort than is needed for stand-alone patterns. Writing an effective pattern language is an incremental process, since it takes a long time to understand a given domain—its scope, problems, and solutions—in sufficient depth. Similarly, pattern languages typically contain many patterns that can also be used as stand-alone patterns, each of which must itself be mature to form the basis for the pattern language. Topology is another factor that drives the incremental maturation of pattern languages: understanding how stand-alone patterns connect in a specific pattern language typically requires extensive experience in building software. Similarly, extending pattern languages with missing patterns is not simply filling empty slots. Patterns and their relationships are not modular, so intensive integration and reworking of the languages and their patterns is often needed."[32]

"Pattern languages can support such a broad solution space because they 'practice' a process of *piecemeal growth*. In particular, by using pattern sequences, a specific solution is developed stepwise through many creative acts until it is complete and consistent in all its parts. Each individual act within the process of piecemeal growth differentiates existing space. A given design or situation

[29] Ibid., 269–70.
[30] Ibid., 137.
[31] Ibid., 291.
[32] Ibid., 292.

in which a particular problem arises is thus gradually transformed into another design or situation in which the problem is resolved by an appropriate pattern of the applied language."[33]

"In particular, each pattern that is applied is integrated into the existing partial design such that it does not violate this design's vision and key properties. Instead, it resolves the problem that arose in accordance with that vision and properties. Applying a pattern is thus structure *transforming*, but vision and property *preserving*. In addition, this stepwise unfolding process avoids architectural drift, because all design activities are governed by the larger structure in which they take place."[34]

"The process of piecemeal growth is also evolutionary, because it supports a stepwise adaptation and refinement of the artifact or 'thing' under development based on its individual needs and requirements. Evolution is achieved through *gradual stiffening*, where developers are 'weaving a structure which starts out globally complete, but flimsy; then gradually making it stiffer but still rather flimsy; and only finally making it completely stiff and strong'."[35]

"When developing software, a mismatch between the current design and the preferred design provides an opportunity for *refactoring*, which involves rewriting a part of a program to improve its structure or readability while preserving its meaning. When applying a pattern language, refactoring involves identifying a misfit pattern and backtracking, so that within the pattern applied prior to selecting the 'wrong' pattern, an alternative is chosen instead. This refactoring process creates—in part—a different path through the pattern language and therefore a different, hopefully more appropriate pattern sequence, according to which the software design can be refactored to meet its requirements."[36]

"When applying a pattern of a pattern language, we recommend first identifying whether elements in the existing structure or design already provide (some of) the roles introduced by the pattern. If so—and if the pattern does not prescribe implementing these roles separately— then let these existing elements keep their roles and extend them with the missing role parts defined by the pattern."[37]

"If, on the other hand, an element of the existing solution already realizes one or more roles introduced by the pattern, but the pattern prescribes implementing these roles separately from each other, refactor the existing solution accordingly."[38]

3.3 Spreading Good Concepts

"Unfortunately, considering patterns as works in progress is a time-intensive process that demands a great amount of effort. This is one reason why there are many more pattern users than pattern authors, and why so many patterns do not escape their place of origin. On the other hand, the return on investment of this time and effort is the reward of the feedback that you receive from the software community and your own increased understanding of the patterns."[39]

[33] Ibid., 298.
[34] Ibid., 299.
[35] Ibid.
[36] Ibid., 303.
[37] Ibid., 309.
[38] Ibid., 311.
[39] Ibid., 60.

“From a practical perspective, good practice poorly communicated is the same as no practice at all: it might as well not exist, and is as mute and unnoticed as any practice communicated poorly. Worse, good practices communicated poorly will lose out to poor practices communicated clearly.”[40]

“if we want to use patterns in our designs and implementations effectively, we must also be able to identify and reference each individual pattern quickly and unambiguously. In addition, even if we do not have particular pattern descriptions or their structure diagrams to hand, we still need a way of talking about these patterns and the designs in which they occur. In other words, we must be able to remember a pattern, otherwise it cannot become a word in our design vocabulary. A pattern that cannot be remembered is less likely to be recalled in practice, regardless of its other qualities. Every pattern therefore needs a name. This name should be evocative. Ideally, if someone references a pattern by its name, anyone familiar with it should be able to recall it from that cue alone. This familiarity is not always easy to achieve. Cute and obtuse names that are meaningful only to a handful of people, such as the clique of the pattern writer, are not necessarily meaningful to others. Recalling that a pattern is a vehicle for communication, a poorly named pattern offers poor transportation. Patterns are most memorable if their names conjure up clear images that convey the essence of their solutions to the target audience.”[41]

“In conversation with pattern authors or proponents, would-be pattern users have the opportunity to question and clarify anything doubtful or unclear. They can also probe details, talk through specific examples, and exchange other thoughts. Through the medium of writing, authors are granted greater precision and control over presentation, but they lose any voice beyond pattern text to assist readers with implementation details, questions of applicability, or clarification of the problem. The dialog gets lost: the audience loses the ability to engage and pattern authors lose the ability to respond. This challenge is one to be overcome in writing a pattern: there still needs to be a sense of dialog. The narrative of a pattern’s documentation must therefore not only express the pattern faithfully and unambiguously, it should also capture some scope of the lost dialog, rather than adopting the stiff character of a monologue. Indeed, if the choice is between a precise but dry account that presents *at* the reader and a slightly less thorough treatment that draws the reader in, pattern authors should favor engagement over disengagement.”[42]

“In addition to all of the other considerations we might cite when favoring one pattern over another, there is a further thought: the selection defines the style of a particular design. That style may be dictated strongly by the language and programming culture, it may a characteristic of a particular framework, or it may be specific to a part of a particular system. Sometimes the adoption of one pattern over another is not so much about resolving a problem as about following a vernacular form. Effective design therefore includes subjective and cultural elements that are neither derivable from first principles nor automatable. At some level the style becomes the substance. For example, Christopher Alexander made many observations about how a collection of one set of patterns helps to define a particular architectural style for buildings that is distinct from other styles drawn from different collections.”[43]

“There are many published examples of how stories have been used to unfold a design or reveal how a situation changed, demonstrating the decisions made and patterns used along the way. Pattern stories can do for a collection of patterns what simpler motivating examples can do for

[40] Ibid., 92.
[41] Ibid., 54–55.
[42] Ibid., 93.
[43] Ibid., 154.

individual patterns: bring them to life and illustrate how they work in practice. The use of such interwoven examples dates back to Christopher Alexander's work in building architecture. For example, a brief, reportage-like example gives an account of the basic progression of a design in *A Pattern Language*. A longer form that includes more rationale can be found in the *The Oregon Experiment*. We also see a tradition of storytelling within software patterns. The aphysical nature of software development inevitably demands the use of examples to make the abstract concrete. In most cases stories are treated as educational and informative extras rather than as something integral to the pattern concept. However, just as we consider the use of examples as a matter of form and not just a formality, we take a similar view of the role played by pattern stories."[44]

"When it comes to sequences, what distinguishes the good from the bad? Certainly, solving the wrong problem qualifies as a problem. For example, a successor pattern may address forces that are not present, or miss those that are. We can also look to the quality of what is built to determine at each stage in the process whether or not the design is in some way whole and stable. A healthy pattern sequence is thus one in which each intermediate step represents a sensible and sufficiently stable structure. While it may not necessarily be a final design, it is not perched precariously in a half-way state that cannot be built or easily described, or that has no obvious follow-on step. A sequence represents a path through a design space, and we would prefer the sequence be based on intermediate steps that are as whole and stable as possible."[45]

Conway's Law: "Any organization that designs a system (defined more broadly here than just information systems) will inevitably produce a design whose structure is a copy of the organization's communication structure."[46]

4. Hands On: Takeaways

1. Conceptual engineering is well-established as an evolving set of best practices for the systems engineering of artifacts at varying levels of abstraction
2. Build a world with a problem, a system, and a story
3. Organizational form matters for success
4. Revision and replacement both occur, but looser organizational structure biases towards the latter
5. Big projects are best carried out by a small team working iteratively for a long time

5. References

Alexander, Christopher, Sara Ishikawa, and Murray Silverstein. *A Pattern Language*. New York: Oxford University Press, 1977.

Beedle, Mike, Martine Devos, Yonat Sharon, Ken Schwaber, and Jeff Sutherland. "SCRUM: An Extension Pattern Language for Hyperproductive Software Development." In *Pattern Languages of Program Design*, 4:637–651, 1999.

Brooks, Frederick P. *The Mythical Man-Month: Essays on Software Engineering*. Reading, Mass: Addison-Wesley, 1975.

Buschmann, Frank, Kevlin Henney, and Douglas C. Schmidt. *On Patterns and Pattern Languages*. Pattern-Oriented Software Architecture 5. Chichester: Wiley, 2007.

Conway, Melvin E. "How Do Committees Invent." *Datamation* 14, no. 4 (1968): 28–31.

[44] Ibid., 189.
[45] Ibid., 195.
[46] Conway, "How Do Committees Invent". Ergo software engineers began to conceptually engineer organizations, e.g. Coplien, "A Development Process Generative Pattern Language"; Beedle et al., "SCRUM"; Coplien and Harrison, *Organizational Patterns of Agile Software Development*.

Coplien, James. “A Development Process Generative Pattern Language.” In *Pattern Languages of Program Design*, 183–237. University of Illinois: Addison-Wesley, 1995.

Coplien, James O., and Neil B. Harrison. *Organizational Patterns of Agile Software Development*. Upper Saddle River, NJ: Prentice Hall, 2004.

Gabriel, Richard. “Foreword.” In *On Patterns and Pattern Languages*, by Frank Buschmann, Kevlin Henney, and Douglas C. Schmidt. Pattern-Oriented Software Architecture 5. Chichester: Wiley, 2007.

Gamma, Erich, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. 1st edition. Reading, Mass: Addison-Wesley Professional, 1994.

Jackson, Daniel. “Conceptual Design: The Essence of Software.” presented at the MIT Professional Education, Accenture India, 2013. https://people.csail.mit.edu/dnj/talks/accenture14/dnj-india-slides-with-animation.pdf.

“Systems Engineering.” In *Wikipedia*, May 3, 2019. https://en.wikipedia.org/w/index.php?title=Systems_engineering&oldid=895344363.

Thompson, Adrian. “Notes on Design Through Artificial Evolution: Opportunities and Algorithms.” In *Adaptive Computing in Design and Manufacture V*, edited by I. C. Parmee, 17–26. London: Springer, 2002. https://doi.org/10.1007/978-0-85729-345-9_2.

Yegge, Steve. “Execution in the Kingdom of Nouns.” *Stevey’s Blog Rants* (blog), March 30, 2006. http://steve-yegge.blogspot.com/2006/03/execution-in-kingdom-of-nouns.html.