Things We Never Told Anyone About Functional Programming

Michael Sperber[1][0000−0002−5280−9632]

Active Group GmbH, Tübingen, Germany
sperber@deinprogramm.de
https://www.deinprogramm.de/sperber/

Abstract. Functional Programming——what is it good for? Programming is the writing of programs. Judging from the papers and most books on functional programming, functional programming excels as a tool for writing short programs——up to a hundred lines maybe. We also know that there are large functional code bases at large companies, and folklore has it that this also works quite well. Yet there is precious little literature on how to go from a hundred lines to a million lines of functional code. This hinders the adoption of functional program by newcomers, who ask questions like:

– How do I find module boundaries?
– How do I organize teams for functional projects?
– How do I onboard newcomers?
– How does functional software development apply to qualities and requirements beyond functionality?

Successful practitioners rely on folklore and experience for the answers to these questions, but precious little is written down in findable places and accessible formats.
This paper is a call to action: The functional programming community needs to do a better job at outreach—find the gaps in our published knowledge, write down what is undocumented and talk to other communities who deal with large-scale software development.

Keywords: Functional Programming · Software Architecture · Domain-Driven Design.

1 Introduction

This paper is about using functional programming to implement software systems, specifically software systems that fulfill externally specified requirements, i.e. requirements that are not inherently about the use of functional programming.

Real-world software systems in active use tend to live long, at least several years. As their maintainers adapt them to new and changed requirements, these systems grow correspondingly. The issues facing such systems are different from the ones facing the small programs typically printed in research papers. These issues have been extensively studied in the field of *software architecture* [34], if mostly from practical angles due to the difficulties of conducting meaningful comparative studies on large software projects.

As software systems grow, changing them tends get more and more expensive per unit of requirements fulfilled, due to a phenomenon called *coupling* [44]: Interdependencies between the parts of a system become more numerous, requiring changes in one part to be accompanied by corresponding changes in other parts, thus "coupling" these parts together.

Consequently, any technique or technology claiming to support the development of such systems must not just support their initial construction, but also their evolution over time. As such, they must grapple with coupling, and support developers in keeping it low or reducing it once it has grown to detrimental levels.

This is particularly important as the requirements imposed on a software systems do not merely concern functionality (i.e. the relationship between input and output), but also other qualities of the system, such as reliability, safety, usability, efficiency etc. [38].

2 Creating Software Systems

Software systems are (presently) created by people, and large software systems are created by large groups of people. This makes the organization of the humans involved in a software project a crucial factor in its success. As the group designing a software system grows, it will typically split into several sub-groups or *teams* to manage the inherent complexity. This raises several questions:

- How should the labor be divided up between the teams?
- What is a good team size?

As to the first question, there are two obvious extreme answers:

- Assign *technical competencies* to each team, i.e. front-end, backend, operations.
- Assign *features* to each team.

("Feature" is a shorthand for a user-visible addition to the system that fulfills requirements running in production.)

The first option typically means that multiple teams have to collaborate to implement a feature. The second option means that each team needs to contain all technical competencies required for the assigned features.

The organizational principle addressing the above questions that is most cited in the software architecture community is *Conway's Law* [8], specifically the following sentence:

> 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.

Concretely, this law (not really a "law" in the colloquial sense) is based on the observation that communication within teams of the organization is more efficient than between teams, and thus will produce designs that have boundaries that correspond to the boundaries between teams. Most organizations seem to have settled on teams not larger than 10 people for the law to hold, with 5–7 being more typical [15].

Consequently, Conway's Law favors organizations that assign features to each team (*feature teams*), as feature teams—in principle at least—can produce features avoiding the overhead of inter-team communication.

Another body of evidence in support of feature teams is the *Accelerate* study [13] that concluded that the following four factors are the most important determinants for the performance of software delivery in an organization:

1. delivery lead time
2. deployment frequency
3. time to restore service
4. change fail rate

The two first factors are slightly surprising: Roughly, they mean that an organization is more effective if it ships features more quickly rather than "more thoroughly" with elaborate processes and sign-off requirements. This also favors feature teams, as they can—again, ideally—deliver features without waiting on other teams. Focusing thus on *velocity* often leads to better results than focusing on the leverage of competency teams, even though it means that any given competency needs to be replicated among many teams.

Note that feature teams are not an all-or-nothing proposition: Most organizations will have certain competencies bundled in teams, with the most recent trend being *platform teams* to manage the complexity of modern cloud platforms [36]. They are a good default, however.

It is easy to imagine a code base that will destroy the advantage of the feature-teams approach through coupling: Conway's Law says that each team will typically manage an identifiable part of the code base. If the parts managed by two teams are strongly coupled, each change one team makes to implement a feature requires a change in the other team's code base, causing overhead and waiting.

So in summary, an effective software organization needs the following:

- connecting requirements and software;
- programming in the small to implement features;
- a software architecture with low coupling when programming in the large and in the long; and
- an organizational structure aligned with the design of system.

3 Programming in the Large and in the Long

The upshot of the previous section was that one of the primary goals of software architecture should be to reduce coupling. This section reviews techniques, successes and failures of the software architecture community to do this.

Software architecture considers coupling mostly between the building blocks of a software, where the term "building block" is loosely defined.[1] Coupling should be low, but of course some coupling is necessary for building blocks to make use of each others' functionality. Within a single software deployment artifact (called *monolith* henceforth), many programming languages provide constructs that make these building blocks explicit, usually called *modules* [3].

The notion of coupling is somewhat problematic in that it refers to the future of the program: Some future features or changes might only require changes in a single building block, indicating low coupling. Others might require changes in multiple building blocks, indicating high coupling. So unless all future changes are known (and they usually are not) any strategy that purports to reduce coupling is necessarily a heuristic.

The study of such heuristics has been around for a long time, often under the heading of *modularity*—low coupling between modules. The best-known example is David Parnas's classic paper [32], which contrasts two different heuristics: One is to establish module boundaries between the steps of a processing pipeline, the other to hide information about design decisions that are difficult or likely to change—with the objective of allowing that decision to be changed later.

Paradoxically, the modern software architecture community sees getting decisions "right" as central [5], which seems to indicate that the field of software architecture has given up on Parnas's original goal. Anecdotally, this is because industrial software projects find it difficult to develop modular software, at least within a monolith. Witness in particular the recent rise of the term "modulith" to refer to modular monoliths, implying that the default monolith is unmodular.

Further testament to the difficulty of modularizing monoliths is the rise of *microservice* architectures since about the end of the 1990s [42]. Such architectures consist of many monoliths that interoperate over the network. One possible benefit of using microservices is horizontal scalability for performance reasons. More often however they are used because they leverage Conway's law to make the development of the system more efficient: Each microservice is under the management of a single team. (One team may manage several microservices, but not the converse.)

As a microservice is (ideally) a self-contained monolith, its development requires little interaction with different teams. In particular, a team does not typically need to coordinate deployment with other teams, making for short delivery lead time and high deployment frequency, the first two tenets of the Accelerate study discussed in the previous section.

One particular style of microservice architecture is *Self-Contained Systems*, where each microservice is a self-contained autonomous web application, sometimes called a *vertical* [35]. The overall system assembles the vertical user interfaces into a (seemingly) coherent whole. The code bases of the different verticals share very little, and communication is mostly through asynchronous feeds, rather than synchronous remote-procedure calls or REST interfaces. The result-

[1] Coupling inside a building block is *cohesion*, where high cohesion is usually desired.

ing system can function even if not all of its verticals are operational, again facilitating the teams developing and releasing independently.

Note that moving from a monolithic architecture to one with microservices entails serious trade-offs: Network communication is significantly slower than intra-monolith calls. Developers need to deal with the vagaries of a distributed system. Whereas communication between the modules of a monolith enjoy the direct support of the programming language, inter-microservice communication needs to address issues such as serialization and choreography. Moreover, comprehensive testing is much more difficult in such a setting.

Microservice architectures make coupling more difficult compared to a monolith, as making use of functionality of a different system is so much work. However, coupling is not impossible and again it is easy to imagine a system organization with high coupling between microservices.[2] Consequently, the question of a good heuristic for placing building-block boundaries remains relevant even in a microservice architecture.

Many large software-development organizations turn to *Domain-Driven Design* (DDD) [11] for such heuristics. The DDD community has developed a rich arsenal of techniques for splitting the domain of a software system into *subdomains*. Subdomain boundaries are usually in places where the vocabulary used by the domain experts shift, and where notions of shared entities change.

For an example, consider the domain of an e-commerce website. (E-commerce tends to be a DDD hotbed.) As users travel through the site, they cross invisible boundaries from one subdomain to the next: On the landing page, they can browse among suggested products, they enter specific searches, look at product details, put products in a shopping cart, and then proceed to checkout. Each of these is typically a different subdomain, and each one has its own notion of "product".

A traditional "enterprise" IT system would have a central database with all information about a product, which would have to be jointly administered throughout the domain, causing high coordination overhead. Splitting into subdomains allows implementing separate building blocks (called *bounded contexts*) for each. Each bounded context can have a separate data model and database for products, enabling efficient specialization and—more importantly—decoupling. Bounded contexts are a good match for a microservice or Self-Contained Systems architecture.

Note that the DDD techniques for identifying subdomains are "only" heuristics with the expectation of low coupling between them. DDD has been quite effective in practice at achieving this goal, but anecdotal evidence suggests that not all DDD-designated boundaries are in exactly the right place. In a microservice architecture, these boundaries are quite difficult to move.

To summarize:

[2] This is sometimes called a "distributed monolith", again implying that the monolith is always unmodular.

– The building blocks of a software system should be loosely coupled to enable effective division of labor and team cooperation.
– Achieving loosely-coupled building blocks in a monolith is difficult in practice.
– Many large software architectures use microservices to establish strong separation of building blocks.
– Moving from a monolith to microservices has negative side effects, but the trade-offs are still often in favor of microservices.
– Domain-Driven Design is a set of techniques for splitting a large domain and system so that development aligns with Conway's law.

4 FP—What Is It Good For?

The previous section made no mention of functional programming: The field of (practically applied) software development and software architecture is dominated by object-oriented programming, and has largely gone untouched by functional techniques. Yet, there are obvious areas where functional programming can make significant contributions to software architecture.

4.1 Coupling and Immutability

Assignment and mutation are among the biggest contributors to coupling, as they create hidden temporal dependencies, which are especially pernicious in the face of concurrency.

While object-oriented programming has acknowledged the advantages of immutability [4], the predominant basic programming model in most projects is still pervasively mutable. Domain-Driven Design makes heavy use of the distinction between (immutable) value objects and (mutable) entities with explicit identity. However, where to actually draw the line between the two is often unclear, and practitioners favor different taxonomies to decide on one or the other.[3]

Moreover, functional data structures [31] are not generally used in object-oriented projects, even though they would fall squarely within the "favor immutability" dictum.

Consequently, many software projects would benefit from a functional style that minimizes mutability and uses functional data structures instead of the traditional mutable collection libraries.

4.2 Data instead of Objects

One way to understand the difference between object-oriented programming and functional programming is that, while object-oriented programming is about

[3] This exposes a deeper issue with object-oriented programming, which is predominantly based on the metaphor of physical objects. Objects inside a computer program are always *representations* of those objects or their state, and this distinction is insufficiently reflected in the object-oriented model.

the physical-objects metaphor, functional programming is about *data*—explicit, immutable representations or descriptions of aspects of our domain.

This distinction is about more than immutability: Many aspects of object-oriented software are not objects. In particular, processes are not, whereas functional programming has a whole taxonomy for representing processes with monads, applicatives, and arrows. Consequently, a possible rallying cry for functional programming is "everything is data".[4]

4.3 Data Modeling

Functional programming has developed a rich discipline of structuring data, as exemplified in the *How to Design Programs* approach to teaching programming [12]. *How to Design Programs* features a taxonomy of data based on the distinction between *sums* and *products* [37]. (Called "itemizations" and "compound data" in the book.) This approach is extraordinarily effective in crafting data models that align with requirements, create good user experiences, and remain flexible in the face of changes.

Surprisingly, object-oriented software development has not developed a similar discipline.[5] Worse, it has largely abandoned object-oriented analysis [14] in favor of ad-hoc designs. In particular, many object models lack a notion of sums, so pervasive in functional programming, especially in languages with algebraic data types.[6]

Of course, functional programming has long gone significantly beyond simple products and sums, in particular with combinator models, in use since at least 1982 [22], and folklore for at least 25 years [33]. Combinator models have applications in many domains [28].

Combinator models solve a frequently occurring problem in software projects, namely that the object/data models (which have degenerated to mere collections of attributes) require modification whenever new requirements arrive or the existing ones change. A combinator model provides a "construction kit" for domain entities rather than a fixed set of monolithic representations, reducing the need to change the software. Since sums are rarely used in object-oriented programming, combinator models are beyond the reach of most real-world projects.

Functional data models also provide a different perspective on bounded contexts: In object-oriented programming, objects and behavior belong inextricably together. Behaviors are often associated with specific bounded contexts. Consequently, this creates pressure on the developers to completely separate data models between different bounded contexts, even if they share meaning and need

[4] It used to be "functions are first-class values", which has become no longer distinctive through the introduction of lambda expressions in most object-oriented languages.

[5] When the DDD community mentions "modeling" (particularly "collaborative modeling"), this typically refers to the modeling of *processes*, not the entities that take part in them.

[6] Of course, sums are expressible as a class hierarchy. However, sums are not a natural outcome from object-oriented analysis. Moreover, creating class hierarchies is largely discouraged in contemporary object-oriented design [17].

to be kept consistent. In functional programming, behavior is typically separate. Moreover, denotational design [9] can establish a common understanding of what pieces of data actually mean.

To summarize, many projects could benefit from the richer, more flexible data models afforded by functional programming.

4.4 Abstraction

Another conspicuous difference between the practice of object-oriented programming and functional programming is the attitude towards abstraction: Current object-oriented practice is generally to avoid abstraction. Anecdotal evidence suggests that abstraction often “goes wrong” in software projects, creating complex object hierarchies. In particular, DDD practitioners commonly advise to stick to the immediate requirements from “the business”, and to avoid adding new terms or functionality beyond them [26].

In contrast functional programming routinely applies abstraction to make software more flexible and gain insights into the structure of the domain.

4.5 Mathematics

Functional programming routinely employs mathematics as a source of inspiration for finding useful abstractions that are applicable in many domains. (Again, explicit use of mathematics is rare in real-world projects.) Furthermore, mathematics can also raise assurance for the consistency and correctness of the software through the use of formal methods.

4.6 Teaching and Systematic Programming

There are many introductory textbooks on functional languages [25,40,1]. Moreover, the *How to Design Programs* approach to teaching programming [12] has revolutionized programming pedagogy by focusing on systematic design using *design recipes*, a set of named techniques that take learners from problem statements to solution programs in easy-to-follow and mostly mechanical steps.

Design recipes have wider implications than just the “teachability” of functional programming: Systematic programming is a useful concept in itself. Systematically constructed programs are less prone to error, and easier to read and manage by others. In object-oriented projects, the equivalent—but much weaker—concept would be the use of a “pattern language”, a project-specific set of rules prescribing similar solutions to recurring problems [27].

Arguably, the more of a software system is constructed systematically, the higher its code quality will be, with positive implications for maintainability and thus its architecture. While there is still room for pushing the envelope on systematic techniques [19], the existing body includes mostly techniques for constructing functional programs.[7]

[7] The second edition of the design recipes book [12] omitted the section on imperative programming, ostensibly because it was not as effective in the classroom as the rest of its material.

5 A Note on Modularity

The previous section contains an obvious omission: Can functional programming contribute to good modularity? It depends, and to examine the issue of modularity, we need to first discuss how to assess "good modularity" realistically. Remember that "good modularity" means low coupling, and low coupling means that typical future changes are local to few modules. Few projects (if any) are sufficiently disciplined to follow Parnas's heuristic for module boundaries.

An effective proxy for a project's modularity is its static dependency graph [27]. The more connected it is, the higher the coupling, and the worse the modularity. Moreover, certain architectural patterns, such as *layers* or *hexagonal architecture* [7] forbid direct dependencies between certain modules.

While there is some indication that functional programs have less dependencies than object-oriented ones [16], excessive dependencies are certainly a problem in real-world functional software systems. Aside from sheer discipline, teams can try to keep dependencies under control by automatically testing the dependency graph for architectural conformance with tools, or enforcing conformance via constraints expressed in a sufficiently expressive module system.

As an example, excessive dependencies are certainly a problem in Java-based projects, which traditionally have a mostly flat namespace. One contributor is, ironically, modern IDEs. These exhaustively index all parts of a project, making any part of a project instantly accessible. This makes it trivial for programmers to create a dependency spanning many modules and potentially violating architectural constraints.

To address this problem, the Java ecosystem features tools such as ArchUnit [2] for unit-testing the dependency graph. Moreover, the module system introduced in Java 9 allows expressing constraints on dependencies, including those from architectural patterns. (That said, many projects make no use of the module system.)

The availability of tools and language features to similarly help reduce dependencies is at best spotty among functional languages. The best candidate may be the OCaml module system [30], which has unfortunately not seen widespread adoption in other functional languages.

6 A Blueprint for Functional Software Design

This section contains a list of tenets of functional software design summarized from the previous sections. These techniques are specific to functional programming yet independent of a particular functional language, and can serve to explain and differentiate software construction using functional programming.

Note that this is just one particular, opinionated list, and others are possible. Specifically, it does not cover the popular techniques using functional programming to implement DDD [41,18], which are different from the techniques originating from the functional programming community.

To summarize, functional design ...

– is bottom-up,
– starts with functional qualities,
– represents everything as data,
– employs data modeling using sums and products,
– embraces abstraction,
– creates combinator models,
– uses algebraic structures, and
– formulates properties mathematically.

Some elaboration of these tenets follows.

Functional design is bottom-up. This statement concerns the construction of software systems, not that of individual functions: Published accounts of practical applications of functional programming usually focus on conceptual modeling of the domain [33,10,23] rather than starting with a macro-architectural template and the filling in the gaps, as is customary in DDD projects.

Functional design starts with functional qualities. Most accounts of functional design describe the implementation of *functionality* rather than a comprehensive plan on how to fulfill requirements on other qualities. The inherent low coupling of functional code allows tackling these requirements later, in line with the order in which they come into focus [21].

Functional design represents everything as data. "Data" (i.e. descriptive, immutable values) here is first of all in opposition to mutable "objects" with state. Moreover, many things are data in functional programs that are not data or objects in object-oriented programs: Functions are data, as well as common abstractions that describe processes or effects. See section 4.2.

Functional design employs data modeling using sums and products. This needs saying as using sums in data modeling is not very common in non-FP projects. See section 4.3.

Functional design embraces abstraction. Abstraction is a key element in functional programming, and present in almost all published accounts of functional programs. See section 4.4.

Functional design creates combinator models. Combinator models are a particular form of abstraction, applied to data modeling, virtually unknown outside of functional programming 4.3.

Functional design uses algebraic structures. Functional design routinely makes use of algebraic structures such as monoids [43], applicative functors [29], arrows [24], or monads [39].

Functional design formulates properties mathematically. This is in opposition to the common practice of specifying behavior through examples. Note that the focus is (currently) on the formulation of properties, not so much their formal verification [6].

7 A Call to Action

Hopefully, this paper has shown that functional programming can make significant contribution to the construction of practical software and their architecture. Unfortunately, this potential has so far largely gone unrealized through the insularity of the functional programming community.

Clearly, we have done too little to communicate techniques for large-scale functional software construction—with the exception of a few books [20,30,41,18], which are mostly specific to particular functional languages.

Consequently, we should engage with other communities—the DDD and software architecture communities in particular. We should not just communicate what FP has to offer, but also listen to the techniques and technologies common in the object-oriented DDD world, and analyze how they could cross-pollinate.

Moreover, the body of knowledge needed to construct large-scale functional projects is far from complete. One particularly pressing question is whether the low coupling afforded by immutability and strong module systems can make splitting out modules as microservices superfluous.

We should more generally investigate how the DDD notion of bounded contexts can translate into functional code, and how functional techniques and language design can help implement them in practical projects. This concerns in particular keeping data models separate between bounded contexts but still consistent. Moreover, it is presently unclear what the best way is to combine the top-down approach of DDD with the bottom-up approach of functional design.

There is much to do. Let's go to work.

References

1. Abraham, I.: F# in Action. Manning (2024)
2. ArchUnit: Unit test your Java architecture. https://www.archunit.org/
3. Barnett, T.O., Constantine, L.L.: Modular Programming: Proceedings of a National Symposium. Information & systems Institute (1968)
4. Bloch, J.: Effective Java. Pearson, 3rd edn. (2017)
5. Brown, S.: Software Architecture for Developers. Leanpub (2022)
6. Claessen, K., Hughes, J.: QuickCheck: a lightweight tool for random testing of Haskell programs. In: Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming. p. 268–279. ICFP '00, Association for Computing Machinery, New York, NY, USA (2000)
7. Cockburn, A., de Paz, J.M.G.: Hexagonal Architecture Explained. Humans and Technology Press (2024)
8. Conway, M.E.: How do committees invent? Datamation (April 1968), https://www.melconway.com/research/committees.html

9. Elliott, C.: Denotational design with type class morphisms (extended version). Tech. Rep. 2009-01, LambdaPix (March 2009), http://conal.net/papers/type-class-morphisms
10. Elliott, C., Hudak, P.: Functional reactive animation. In: Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming. Association for Computing Machinery, New York, NY, USA (1997)
11. Evans, E.: Domain-Driven Design: Tackling Complexity in the Heart of Software. Addison-Wesley (2003)
12. Felleisen, M., Findler, R.B., Flatt, M., Krishnamurthi, S.: How to Design Programs. MIT Press, second edn. (2018)
13. Forsgren, N., Humble, J., Kim, G.: Accelerate: The Science of Lean Software and DevOps: Building and Scaling High Performing Technology Organizations. IT Revolution Press (March 2018)
14. Fowler, M.: Analysis Patterns: Reusable Object Models. Addison-Wesley Professional (1996)
15. Fowler, M.: Two pizza team. https://martinfowler.com/bliki/TwoPizzaTeam.html (Jul 2023)
16. Gabasova, E.: Comparing F# and C# with dependency networks. https://evelinag.com/blog/2014/06-09-comparing-dependency-networks/ (June 2014)
17. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns. Elements of Reusable Object-Oriented Software. Prentice-Hall (1994)
18. Ghosh, D.: Functional and Reactive Domain Modeling. Manning (2016)
19. Gibbons, J.: How to design co-programs. Journal of Functional Programming **31** (2021)
20. Granin, A.: Functional Design and Architecture: Examples in Haskell. Manning (2024)
21. Harrer, M.: Quality Tactics. Leanpub (2024)
22. Henderson, P.: Functional geometry. In: Proceedings of the 1982 ACM Symposium on LISP and Functional Programming. pp. 179—187. LFP '82, Association for Computing Machinery, New York, NY, USA (1982)
23. Hudak, P.: The Haskell School of Expression: Learning Functional Programming through Multimedia. Cambridge University Press, 4th edn. (2000)
24. Hughes, J.: Generalising monads to arrows. Sci. Comput. Program. **37**(1–3), 67–111 (May 2000)
25. Hutton, G.: Programming in Haskell. Cambridge University Press, 2nd edn. (2016)
26. Khononov, V.: Learning Domain-Driven Design: Aligning Software Architecture and Business Strategy. O'Reilly (2021)
27. Lilienthal, C.: Langlebige Softwarearchitekturen. Dpunkt Verlag, 4th edn. (2024)
28. Maguire, S.: Algebra-Driven Design. Leanpub (2020)
29. McBride, C., Paterson, R.: Applicative programming with effects. J. Funct. Program. **18**(1), 1–13 (Jan 2008)
30. Minsky, Y., Madhavapeddy, A.: Real World OCaml: Functional Programming for the Masses. Cambridge University Press, 2nd edn. (2022)
31. Okasaki, C.: Functional Data Structures. Cambridge University Press (2008)
32. Parnas, D.L.: On the criteria to be used in decomposing systems into modules. Commun. ACM **15**(12), 1053–1058 (Dec 1972)
33. Peyton Jones, S., Eber, J.M., Seward, J.: Composing contracts: An adventure in financial engineering (functional pearl). In: Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming. p. 280–292. ICFP '00, Association for Computing Machinery, New York, NY, USA (2000)

34. Richards, M., Ford, N.: Fundamentals of Software Architecture: An Engineering Approach. O'Reilly (2020)
35. Self-contained systems (SCS). https://scs-architecture.org/
36. Skelton, M., Pais, M.: Team Topologies. IT Revolution Press (2019)
37. Sperber, M., Wehr, S.: Data modeling with sums and products. https://funktionale-programmierung.de/2024/11/25/sums-products-english.html (November 2024)
38. Starke, G., et al.: arc42 quality model. https://quality.arc42.org/
39. Wadler, P.: The essence of functional programming. In: Proceedings of the 19th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. p. 1–14. POPL '92, Association for Computing Machinery, New York, NY, USA (1992)
40. Whitington, J.: OCaml from the Very Beginning. Coherent Press (2013)
41. Wlaschin, S.: Domain Modeling Made Functional. O'Reilly Media (2018)
42. Wolff, E.: Microservices: Flexible Software Architecture. Addison-Wesley Professional (2016)
43. Yorgey, B.A.: Monoids: theme and variations (functional pearl). In: Proceedings of the 2012 Haskell Symposium. p. 105–116. Haskell '12, Association for Computing Machinery, New York, NY, USA (2012)
44. Yourdon, E., Constantine, L.L.: Structured Design. Prentice-Hall (1979)