Object Disorientation

Marvin Wyrich, *Saarland University, Saarland Informatics Campus, Germany*
Johannes C. Hofmeister, *Heidelberg University, Germany*
Sven Apel, *Saarland University, Saarland Informatics Campus, Germany*
Janet Siegmund, *Chemnitz University of Technology, Germany*

Abstract—Despite its longstanding history, object-oriented programming (OOP) remains a cornerstone of modern software development. While everyone seems to have an idea of what OOP means, it is surprisingly challenging to operationalize key concepts of OOP due to its widespread but varied perceptions. This makes it difficult for researchers to investigate the paradigm, difficult for practitioners to select suitable technologies, and complicates students' transfer of OOP knowledge across programming languages. Our study aimed to uncover the essence of OOP—its fundamental ideas, concepts, and principles—through a scoping review of prominent views in both academic publications and non-academic sources, such as industry reports or practitioner blogs. In the end, we synthesized two conceptual lines within OOP: understanding "why" and "how" the paradigm is used. Our synthesis reveals they are complementary yet often misunderstood as separate perspectives. Addressing this disconnect is crucial for fostering clearer communication and effective use of OOP in practice.

Chances are good that most readers have heard of *object-oriented programming* (OOP). After all, the paradigm predates many of the programming languages currently in use. And even if OOP may no longer be a trendy topic discussed in tech magazines, it remains a significant part of everyday life for many developers. For example, most modern programming languages provide object-oriented features, such as objects, interfaces, classes, and inheritance. In 2013, Jonathan Aldrich even went so far to call objects "inevitable" [1].

Proponents of OOP often claim that the reason for OOP's success is best explained by its intuitiveness or naturalness [2], because one would think in terms of categories, much like one would categorize animals or plants [3]. While OOP is a dominant paradigm, not everybody agrees that OOP is actually the best way to write programs. Some developers may say that OOP can quickly lead to over-engineered solutions, requires a lot of distracting, unnecessary code to describe simple concepts, and thus may lead to complicated solutions that are big, bloated, and difficult to maintain.

Digital Object Identifier 10.1109/XXX.0000.0000000

Given these differing perspectives, it becomes important to take a scientific approach to the matter and empirically investigate the positive and negative effects of using different paradigms. The only problem is that we do not really know what distinguishes object-oriented code from code that follows a different paradigm. If we wanted to compare, for example, how the approach of a developer differs between OOP and a different programming paradigm, we would fail due to a lack of clear distinction between them.

In the 1970s, object-oriented ideas grew quickly and were adopted by many programming-language designers. Yet, some were critical of this fast growth: Despite OOP's success story, in 1982, ten years after the term "object-oriented programming" first appeared in the context of Smalltalk 72, Tim Rentsch expressed his confusion: "What is object oriented programming? My guess is that object oriented programming will be in the 1980's what structured programming was in the 1970's. Everyone will be in favor of it. Every manufacturer will promote its products as supporting it. Every manager will pay lip service to it. Every programmer will practice it (differently). And no one will know just what it is" [4].

Today, the term "object-oriented" can indeed be found in many areas of programming and software en-

gineering. Mainstream programming languages, such as Java, C#, C++, JavaScript, Python, and Rust, support object-oriented features, and so do languages dedicated to modeling and analysis (e.g., UML and Alloy). The object-oriented paradigm has influenced both research and practice: Researchers have been exploring the ideas behind the paradigm, such as its formal semantics, or metrics to estimate development costs. Software engineers are routinely using object-oriented languages for analysis, design, and implementation. And students learn about OOP in programming courses. In short, the object-oriented paradigm is unavoidable and affects large parts of a programmer's reality.

Yet, the programming community is still as confused by different notions of anything "object-oriented" as Rentsch was over 40 years ago. It looks like Rentsch's statement still holds true today: There are many interpretations of the object-oriented paradigm, but there is no canonical agreement on what it means precisely. This may have been advantageous for the successful spread of the abstract concept that OOP is, but this disagreement also leads to some concrete challenges.

1) Researchers cannot meaningfully investigate the subject. Without a clear definition of what object orientation is, there is no (easy) way to distinguish it from other paradigms. This renders the subject difficult to research and makes it almost impossible to demonstrate its value as a whole or in part.
2) Practitioners struggle to assess and select technologies, such as specific programming languages or frameworks, in an informed manner. Since software supports businesses in achieving their economic goals, any new technology must be evaluated to estimate its potential impact on costs. Without a clear definition of OOP, it is impossible to accurately evaluate the costs associated with a technology, such as employee training expenses.
3) Students who are learning to program in an object-oriented language may encounter difficulties when transferring object-oriented concepts from one language to another because it may not be clear what the object-oriented concepts are or how they work, independent of the programming language.

When there is no clarity on what we mean when we use the term "object-oriented", we may risk talking past each other, potentially slowing down research, practice, and learning efforts. If there was a clear definition of OOP, implementation details would not be open to the interpretation of programming-language designers, and informed choices could be made about technologies. In this essay, we go to the bottom of the diversity of notions and concepts of object orientation, specifically focusing on its role within programming languages, and discuss implications for researchers, practitioners, and students to improve the current situation of "object disorientation". While object orientation is also applied in other areas of software engineering, such as requirement analysis, we limit our discussion here to its interpretation within the context of programming languages. Specifically, we are looking for an answer to the following research question: **What are the differences and similarities between authoritative definitions of the *object-oriented* programming paradigm?**

SEARCHING DEFINITIONS: A SCOPING REVIEW

To address our research question, we conducted a *scoping review* to create a thorough understanding of object orientation. Such scoping review is a form of systematic literature review "to understand the status of research on a particular topic, typically by mapping primary studies into categories" [5].

Our search strategy was a pure *snowballing* approach, motivated by evidence that snowballing is similarly effective as and usually more efficient than database searches. As the starting point for our scoping review, we selected Aldrich's Onward! 2013 essay "The Power of Interoperability: Why Objects Are Inevitable" [1]. We started with this essay because we kept coming back to this article in discussions about object orientation with members of our groups and beyond. Furthermore, this article is often mentioned in blogs about object orientation.[1] This essay led to some highly cited papers and eventually to the 24 publications that we included in our scoping review. At this point, we have extracted the similarities and differences in understanding object-oriented programming, and have reached information saturation in consideration of our research question.

TWO PERSPECTIVES ON OBJECT ORIENTATION

During our search for authoritative definitions of *object-oriented*, we discovered that many different interpretations exist. They can be roughly grouped along two conceptual lines, which West calls the *formalist*

[1] Just to mention two:
http://lambda-the-ultimate.org/node/4790/
https://www.bcobb.net/hacker-school-read-along-the-power-of-interoperability/

and the *hermeneutic* interpretation [6]. Hermeneutics, broadly defined as the theory of interpretation and comprehending the world, focuses on understanding why certain concepts are used and how they relate to human needs and contexts. Applied to the subject at hand, the hermeneutic interpretation of OOP emphasizes understanding the rationale behind using objects—such as reuse, modularity, or problem decomposition—and how these properties align with the broader goal of managing complexity in programs. By contrast, "formalistic" in linguistics refers to a set of production rules, and within this frame, objects and object-oriented systems are discussed in terms of their technical features. The formalist view strives for correctness and verifiability, whereas the hermeneutic view strives for meaning and plausibility.

We start discussing the differences between the formalist and hermeneutic perspective with a quote from Jonathan Aldrich: "The key design leverage provided by objects is the ability to define nontrivial abstractions that are modularly extensible, where instances of those extensions can interoperate in a first-class way" [1].

In his essay, Aldrich argues for the empirical success of objects and object-oriented programming. He raises the question: Given that objects are essentially procedural data structures, what explains their evident success today? Aldrich reasons that objects are extensible abstractions, which he calls *service abstractions*.

Aldrich builds on the work of Cook, which contrasts objects with abstract data types (ADTs) [7]. ADTs provide an implementation-independent definition of a data type, specifying its behavior through the operations it supports. Objects and ADTs are not the same, and while objects are more extensible than ADTs, ADTs are easier to verify than objects. He points out that most modern programming languages provide a mixture of ADTs and objects, with some using classes as static types, while others use classes as object factories and use interfaces as static types instead.

Cook points back to Cardelli and Wegner [8] and outlines that their article "On Understanding Types, Data Abstraction, and Polymorphism" initiated a lot of research on the semantics of object-oriented programming. The astonishing number of more than 3000 citations supports this claim. Aldrich, Cook, as well as Cardelli and Wegner discuss formal aspects of OOP, and describe OOP as the sum of three parts ("object-oriented = data abstractions + object types + type inheritance" [8]), thus adopting a formalist perspective.

The hermeneutic perspective is represented by Alan Kay, who is considered the inventor of the term object-oriented. Kay's essay on "The Early History of Smalltalk" is cited by Aldrich [1], but fragments of Kay's ideas are commonly mentioned in popular sources over the Web and point to e-mails that have been sent to various individual recipients and to mailing lists ([9], [10], [11]). Kay's ideas are in stark contrast to the ideas of the formalist line, as the following quote from his essay demonstrates: "Smalltalk's design – and existence – is due to the insight that everything we describe can be represented by the recursive composition of a single kind of behavioral building block that hides its combination of state and process inside itself and can be dealt with only through the exchange of messages" [10].

Kay's ideas are associated with the hermeneutic view, while Aldrich, Cook, as well as Cardelli and Wegner are prototypical for formalist views[2]. The formalist line focuses on technical features that define or categorize what object-oriented programming is and how it can be implemented (e.g., classes, inheritance), whereas the hermeneutic line emphasizes the problems that these features are addressing (e.g., improve understanding and maintenance efforts). In other words, the first line focuses on *how* to apply the object-oriented paradigm, while the second focuses on *why* it should be applied.

HOW AND WHY IS OBJECT ORIENTATION USED?

For a brief overview, we have summarized the two conceptual views with their respective focus on specific components in Figure 1. Our basic assumption is that the formalist view and the hermeneutic view are two sides of the same coin, that is, both sides describe the same abstract concept of object-oriented programming, but focus on different aspects.

[2]Note that, at least, Aldrich's work [1], while rooted in formalist definitions, connects the technical characteristics of objects to their benefits, illustrating a bridge between the two perspectives and underscoring their interdependence. We will come back to this aspect in a moment in our basic assumption that the two views represent two sides of the same coin.

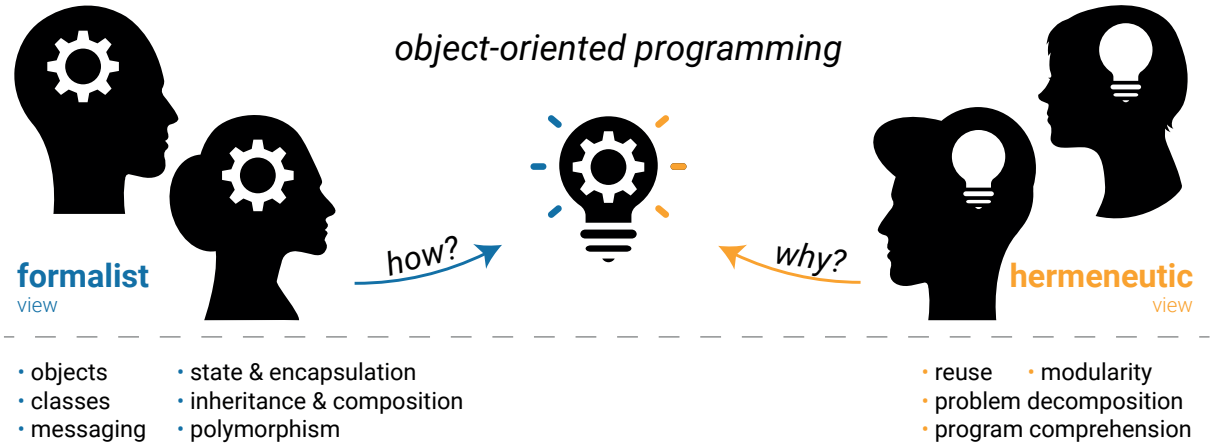

FIGURE 1. Overview of the components of the two conceptual lines (*how* & *why*) for the definition of object-oriented programming resulting from our scoping review

The formalist view (*how* OOP is applied) deals with:

- **Objects:** Objects as units of conception to reduce complexity and support programmers to describe programs
- **State & Encapsulation**: The idea of combining data and operations to constitute objects with (mutable) state, and retaining the data within the boundaries of the object
- **Classes:** Classes as a means of describing and creating similar objects according to a metaphorical blueprint
- **Inheritance & Composition:** Building hierarchies of classes, and hierarchies of objects
- **Messaging:** Invoking operations by sending messages to objects that themselves are responsible for performing an action; messages are (dynamically) dispatched to the corresponding objects based on their types
- **Polymorphism:** One object can be replaced by another, as both can respond to the same set of messages, even if in different ways (e.g. a rectangle and a circle object could both react to a *draw* message); based on the receiver type, messages are dynamically dispatched to the corresponding objects

Thus, from a formalist view, we could summarize OOP in a technical sense as objects, which have a state, are defined in or grouped by classes, can take different forms and send each other messages. By contrast, the hermeneutic view discusses the following reasons as to *why* OOP is applied:

- **Reuse:** Applying objects in the same or a different context
- **Modularity:** Defining boundaries to facilitate local reasoning
- **Problem Decomposition:** Identifying the operands of the problem-solving process, and building corresponding representations of these operands to design a solution
- **Program Comprehension:** Supporting programmers in understanding, navigating, and reasoning about the program by facilitating mental processing and problem-solving

Thus, from a hermeneutic point of view, we could summarize OOP as a programming paradigm that allows for reuse, modularization, problem decomposition, and program comprehension.

In our accompanying technical report[3], we discuss all of these categories in detail based on the selected papers. There, we dive deeper into the how's and why's of OOP and the way they shape our today's (differing) understanding of the term. As we have already indicated, we do not regard the two views as mutually exclusive. Rather, for example, several object-oriented mechanisms from the formalist view facilitate the aspects of the hermeneutic view in the first place. The mechanisms of polymorphism, inheritance, composition, and encapsulation, for example, enable reuse. What we find worthy of discussion here instead is that there is not only the theoretical separation into

[3]https://doi.org/10.5281/zenodo.14699662

two conceptual lines, but there are also many different angles within the two perspectives themselves, which we discuss next.

TWO PERSPECTIVES, MANY ANGLES

The hermeneutic view is a view about people and focuses on supporting people in decomposing problems, encoding them as computer programs, and comprehending existing programs (i.e., a view about *why* OOP should be used). By contrast, the formalist view concentrates on mechanical aspects, modeled in form of formal semantics and object calculi, and is mainly concerned with the static properties of object-oriented features, such as objects, classes, and their inner workings (i.e., a view about *how* OOP works and which concepts make up OOP).

The two perspectives in themselves represent very different points of view, but when we investigated the differences and similarities between the definitions, we found further disagreement about the significance for OOP of some of the components of the respective view. For example, some authors conclude that *state* is not essential to define objects (e.g., [7], [12]), while others disagree (e.g., [3], [13]) and define objects as entities having state. A similar situation applies to the concept of *classes*. Wegner [13], for example, describes classes as a primary means to create objects, and regards them as essential to characterizing *object-oriented* programming. Kay, however, underscores that classes are not the central idea of Smalltalk [11], although Smalltalk uses classes extensively. Therefore, some see classes as fundamental to defining OOP, whereas others do not.

The interpretation of polymorphism within object orientation is generally consistent, yet the term itself can cause confusion due to its broader use in programming. Thomas describes polymorphism as “the ability of different objects to respond differently to the same message” [14], highlighting that objects can be substituted for each other as long as they respond to the same set of messages. Pierce [15] adds nuance by distinguishing between types of polymorphism, including parametric polymorphism, which allows general implementations independent of concrete types, and subtype polymorphism, which supports refining types. These different kinds of polymorphism are rarely kept apart, leading to confusion across communities: “The unqualified term ‘polymorphism’ causes a certain amount of confusion between programming-languages communities. Among functional programmers (i.e., those who use or design languages such as ML, Haskell, etc.), it almost always refers to parametric polymorphism. Among object-oriented programmers, by contrast, it almost always means subtype polymorphism, while the term *genericity* (or *generics*) is used for parametric polymorphism” [15].

Discussions and controversies about the details of the components of the object-oriented paradigm are somehow limited to the elements of the formalist view, though. We find much more consensus within the hermeneutic view. For example, there is agreement among various authors that objects and classes to reuse code are very beneficial (e.g., [12], [14]). Controversies are then essentially limited to the pragmatic specifics of how exactly reuse best is achieved, for example through concepts that we attribute to the formalist view, such as encapsulation and inheritance.

Now, one could critically question whether the conceptual elements of the hermeneutic view are perhaps simply too abstract to cause controversy. Some readers may even think that parts of the hermeneutic view are not even specific to OOP, but could also apply in this form to other programming paradigms. This is certainly a valid concern. Nevertheless, the two views together represent exactly what is strongly attributed (at least) to the object-oriented paradigm in the literature.

To summarize, there is a common ground of views on OOP, but there is also considerable disagreement on the specifics. Thus, the different views do not only play out on the abstract level, where we distinguish between the how and why; the details of the how and the details of why also offer potential for discussion among the authors of the literature we found. This can potentially lead to confusion.

In the next section, we discuss potential causes for this situation and how we should deal with the implications of a multifaceted meaning of OOP to overcome some of the confusion that motivated this essay.

FROM OBJECT DISORIENTATION TO OBJECT ORIENTATION

It is in the nature of abstract concepts that different people think of them in (slightly) different ways. OOP is one such concept, and we have observed that different people do indeed associate very different aspects with it, such that the importance and usefulness of specific object-oriented features is challenged by different authoritative voices. It appears that they all favor OOP, but in different ways.

OOP can be understood as a phenomenon whose success is probably not due to its technical superiority over other paradigms. Instead, its success may stem from its adaptability and the way it resonated

with the interests and needs of diverse individuals. Various people contributed to, created, and discussed notions of the object-oriented paradigm as a vague idea rather than a formal definition of the concept. Thus, we hypothesize that, just like in a game of telephone, the content of what was shared changed and evolved. At the very least, different people could identify with and interpret it in ways that suited their own contexts, enabling them to build upon it without being constrained by a strict formal definition.

There appears to be no consensus on what *object-oriented programming* means, and different people may have different views on what they understand by OOP. More importantly, there may not even have to be a consensus. However, problems arise when people are not aware of these differences, do not or cannot articulate their own views, and therefore talk past each other. This causes various problems, for example, for researchers (like us) trying to pinpoint how OOP drives the cognitive processes during program comprehension, practitioners trying to evaluate the usefulness of a tool, or educators compiling a syllabus.

With our literature review and the resulting summary of the different views in Figure 1, we seek to counteract this problem by revealing the multifaceted nature of OOP systematically for the first time. Based on our findings, the following five recommendations aim to address challenges faced by various groups engaging with OOP, including researchers, practitioners, and educators. While each recommendation has a distinct focus, together they highlight the importance of clarity, exploration, and perspective when engaging with OOP.

> RECOMMENDATION 1: Everybody, be aware of the formalist and hermeneutic perspectives.

OOP is not just programming with a set of language features, but might require a certain mindset about the benefits of these very features. On the one hand, reuse, modularity, problem decomposition, and program comprehension mean reflecting on the world and finding appropriate abstractions to deal with inherent complexity. This is an abstract view of *why* OOP can be beneficial. On the other hand, without technical concepts such as objects, classes, state and inheritance, many programmers might not be able to achieve these benefits as effectively or even think of the object-oriented paradigm. Certain specific features are therefore also part of OOP, that is, a perspective on *how* OOP is used. Both views are not mutually exclusive, but complement each other. Although the object-oriented programming paradigm appears—in principle, at least—to be able to unite the formalist and hermeneutic perspectives, being unaware of the two parts might lead people to mistaking one for the other.

> RECOMMENDATION 2: Researchers and educators, explicitly distinguish between perspectives on OOP.

Citing sources is standard practice in scientific research. However, it is often omitted for widely recognized and common knowledge. For instance, software researchers typically do not cite well-known programming languages, much like social scientists often omit citations for commonly used statistical tests. Knowledge about these tools is simply assumed. However, since knowledge in the context of OOP differs to a large extent, it is important to make more explicit reference to what is meant by the broad concept of OOP. This counts particularly for researchers whose task is to communicate their research intentions as precisely as possible. But it is also important for those who seek to educate others about OOP and shall thereby convey that there is more than one perspective on the matter. Textbooks on object-oriented programming should differentiate the paradigm from others more explicitly in this respect than is currently the case. The formalist and hermeneutic views are usually implicitly mixed in the form of pragmatic tips on program design and the discussion of the advantages of such a design, without explicitly conveying to the reader what view of OOP is assumed on a conceptual level.

> RECOMMENDATION 3: Researchers and practitioners, do not mistake the parts for the whole

There are numerous alternative interpretations of what features and concepts belong to OOP, not only when reading and writing scientific papers about it, but also when debating programming styles with a developer colleague from the neighboring office. The sum of these features and concepts makes up OOP.

For instance, a case study analyzing inheritance in an object-oriented program might conclude that OOP is unsuitable for large software systems due to inheritance introducing coupling, breaking encapsulation, and resulting in difficult-to-maintain code. While these conclusions may be valid in specific cases, we contend that such reasoning could represent an attribution error. Concluding that OOP is “bad” because inheritance caused problems is only valid if inheritance is understood as an essential part of OOP. If the maintenance of a system is difficult because of inheritance, the intuitive conclusion should be that there

is a problem with inheritance or system design, not with OOP. Clearly, OOP extends beyond mere use of classes, objects, and inheritance. It involves analyzing problems and structuring solutions in terms of objects, which can significantly influence software design, for better or worse. This holistic approach to problem-solving encapsulates the essence of OOP.

Another example is that researchers studying a programmer's ability to understand object-oriented code should carefully define what they mean by "object-oriented" in their experiments. Instead of creating code that assumes a universal understanding of OOP—given the varied interpretations of what OOP entails—it is more beneficial to develop code based on an explicitly defined subset of OOP concepts and features. This approach aligns with the recommendations to clearly define our terms (Rec. 2) and not mistake the parts for the whole (Rec. 3). While creating code that unambiguously represents all aspects of OOP may be impractical, researchers can mitigate ambiguity by being explicit about which aspects of OOP their code represents.

> RECOMMENDATION 4: Practitioners, explore the breadth of object orientation.

Understanding the various aspects of the object-oriented programming paradigm and recognizing the debate surrounding its features provides flexibility for adopters, allowing for the adoption of alternative perspectives. For instance, users of class-based languages might explore prototype-based languages. The idea that classes are not essential and OOP can be achieved without them could reveal new possibilities.

Another example is the long-standing awareness among programmers that global, mutable state can complicate reasoning about a program's behavior and should be avoided. Encapsulation promotes restricting access to state from the outside, potentially leading to more maintainable code. If implementing this proves challenging, it may be beneficial to explore programming language systems that conceptually avoid mutable state, as seen in many functional languages such as Haskell or F#.

> RECOMMENDATION 5: Researchers, object-oriented programming is not the same as Java.

We should critically reflect on how computer science and programming language researchers may oversimplify the object-oriented paradigm: Research on object-oriented programming sometimes focuses on specific languages such as Java (or another purportedly object-oriented language). It is essential not to conflate the general OOP paradigm with specific implementations. Each programming language and its ecosystem may demand, require, or promote a particular mindset when working with objects, and this approach may not always be transferable to other programming languages.

Conclusion

We set out to find a definition of object-oriented programming that reflects how the majority of researchers and practitioners think about this programming paradigm. We intended to provide a common ground for people interested in the object-oriented paradigm to communicate their requirements efficiently without talking past each other. By comparing formalist and hermeneutic views, along with their related components, we provide a clearer understanding of what might be encompassed by OOP. Our five recommendations should further enhance clarity in the future. Therefore, we believe that we have advanced towards our goal of offering a compass and map for researchers, practitioners, educators, and students to navigate the multifaceted world of the OOP paradigm. We conclude that there is no singular truth about objects; thus, when discussing anything object-oriented, the first question should always be: What exactly do you mean?

ACKNOWLEDGMENTS

We thank Chris Parnin and Christian Kästner for comments on an early draft. This work has been supported by the DFG Grants AP 206/6, AP 206/14, and SI 2045/2-2, by Grant 389792660 as part of Transregio Collaborative Research Center 248 – CPEC, as well as by ERC Advanced Grant "Brains On Code" (101052182).

REFERENCES

1. J. Aldrich, "The Power of Interoperability: Why Objects are Inevitable," in *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*. ACM Press, 2013, pp. 101–116.
2. D. Robson, "Object-oriented software systems," *Byte*, vol. 6, no. 8, pp. 74–86, 1981.
3. G. Booch, R. A. Maksimchuk, M. W. Engle, B. J. Young, and J. Conallen, *Object-Oriented Analysis and Design with Applications*, 3rd ed. Boston, MA: Addison-Wesley, 2007.

4. T. Rentsch, “Object Oriented Programming,” *ACM SIGPLAN Notices*, vol. 17, no. 9, pp. 51–57, 1982.
5. P. Ralph and S. Baltes, “Paving the way for mature secondary research: The seven types of literature review,” in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2022. New York, NY, USA: Association for Computing Machinery, 2022, p. 1632–1636. [Online]. Available: https://doi.org/10.1145/3540250.3560877
6. D. West, *Object Thinking*. Redmond, WA: Microsoft Press, 2004.
7. W. R. Cook, “On Understanding Data Abstraction, Revisited,” *ACM SIGPLAN Notices*, vol. 44, no. 10, pp. 557–572, 2009.
8. L. Cardelli and P. Wegner, “On Understanding Types, Data Abstraction, and Polymorphism,” *ACM Computing Surveys.*, vol. 17, no. 4, pp. 471–522, 1985.
9. A. Kay and S. Ram, “Re: Clarification of "Object-Oriented"," 2003. [Online]. Available: http://www.purl.org/stefan_ram/pub/doc_kay_oop_en
10. A. Kay, “The Early History of Smalltalk,” in *Proc. Conf. on History of Programming Languages (HOPL-II)*. ACM Press, 1993, pp. 69–95.
11. ——, “Prototypes vs. Classes Was: Re: Sun’s HotSpot,” 1998. [Online]. Available: http://lists.squeakfoundation.org/pipermail/squeak-dev/1998-October/017019.html
12. A. Snyder, “The Essence of Objects: Concepts and Terms,” *IEEE Software*, vol. 10, no. 1, pp. 31–42, 1993.
13. P. Wegner, *Dimensions of Object-based Language Design*. New York, NY, USA: ACM Press, 1987, vol. 22, no. 12.
14. D. Thomas, “What’s in an Object?” *Byte*, no. March 1989, pp. 231 – 240, 1989.
15. B. C. Pierce, *Types and Programming Languages*. Cambridge, MA: MIT Press, 2002.

Marvin Wyrich is a postdoctoral researcher at the Saarland University, where he has been part of the Chair of Software Engineering since 2023. He received his Ph.D. from the University of Stuttgart in 2023. His research interests include empirical and behavioral software engineering, with a focus on program comprehension, science communication, and developing sound research methodologies. Contact him at wyrich@cs.uni-saarland.de

Johannes C. Hofmeister works as a software developer, system administrator, and author at the University of Heidelberg, Germany. In his work, he focusses on how people express themselves when programming. Contact him at johannes.hofmeister@psychologie.uni-heidelberg.de.

Sven Apel holds the Chair of Software Engineering at Saarland University & Saarland Informatics Campus, Germany. Prof. Apel received a Ph.D. in Computer Science in 2007 from the University of Magdeburg. His research interests include the development and evaluation of methods, tools, and theories for the construction and analysis of efficient, reliable, maintainable, and configurable software systems. In this endeavor, he pays special attention to the human factor and interdisciplinary research questions. Contact him at apel@cs.uni-saarland.de.

Janet Siegmund is professor for Software Engineering. Before, she led the junior research group PICCARD, funded by the Centre Digitisation.Bavaria. She received her Ph.D. from the University of Magdeburg in 2012 and holds two master’s degrees (Computer Science and Psychology). Her research is centered around the human factor in software engineering. She regularly serves as program-committee member or chair for conferences and workshops and was in the steering committee of the International Conference on Program Comprehension. Her work received several distinguished paper awards (ICSE, FSE, SANER) and two Most Influential Paper Awards (ICPC, GPCE). Contact her at janet.siegmund@informatik.tu-chemnitz.de.