



# **20. Workshop Software-Reengineering und -Evolution**

der GI-Fachgruppe Software-Reengineering (SRE)

## **9. Workshop „Design for Future“**

des GI-Arbeitskreises „Langlebige Softwaresysteme“ (L2S2)

**Bad-Honnef**

**02.–04. Mai 2018**



# 20. Workshop Software-Reengineering und -Evolution der GI-Fachgruppe Software-Reengineering (SRE)

## 9. Workshop „Design for Future 2018“ des GI-Arbeitskreises „Langlebige Softwaresysteme“ (L2S2)

02.-04. Mai 2018  
Physikzentrum Bad Honnef

Die Workshops Software-Reengineering (WSR) im Physikzentrum Bad Honnef wurden mit dem ersten WSR 1999 von Jürgen Ebert und Franz Lehner ins Leben gerufen, um neben den internationalen erfolgreichen Tagungen im Bereich Reengineering (wie etwa WCRE und CSMR) auch ein deutschsprachiges Diskussionsforum zu schaffen. Seit dem Jahr 2014 firmiert der Workshop unter dem Titel „Software-Reengineering & -Evolution“ (WSRE), um eine breitere Zielgruppe anzusprechen und auf den Workshop aufmerksam zu machen.

Ziel der Treffen ist es nach wie vor, einander kennen zu lernen und auf diesem Wege eine direkte Basis der Kooperation zu schaffen, so dass das Themengebiet eine weitere Konsolidierung und Weiterentwicklung erfährt.

Durch aktive und gewachsene Beteiligung vieler Forscher und Praktiker hat sich der WSRE als zentrale Reengineering-Konferenz im deutschsprachigen Raum etabliert. Dabei wird er weiterhin als Low-Cost-Workshop ohne eigenes Budget durchgeführt. Bitte tragen auch Sie dazu bei, den WSRE weiterhin erfolgreich zu machen, indem Sie interessierte Kollegen und Bekannte darauf hinweisen.

Auf Basis der erfolgreichen WSR-Treffen der ersten Jahre wurde 2004 die GI-Fachgruppe Software-Reengineering gegründet, die unter <http://www.fg-sre.gi-ev.de/> präsent ist. Durch die Fachgruppe wurden seitdem neben dem WSRE auch verwandte Tagungen zu Spezialthemen organisiert. Seit 2010 ist der Arbeitskreis Langlebige Softwaresysteme (L2S2) mit seinen „Design For Future“-Workshops (DFF) aufgrund der inhaltlichen Nähe ebenfalls bei der Fachgruppe Reengineering aufgehängt. In der Regel alle zwei Jahre findet seitdem ein gemeinsamer Workshop von WSRE und DFF statt, auf Grund des WSRE-Jubiläums auch in diesem Jahr, bereits nach einjährigem Abstand. Diese Kombination soll den Austausch zwischen den beiden Gruppen fördern. Während beim DFF der Schwerpunkt auf wartbaren Architekturen liegt, widmet sich der WSRE weiterhin den allgemeinen Themen Reengineering und Evolution in all ihren Facetten.

Der WSRE ist die zentrale Tagungsreihe der Fachgruppe Software-Reengineering. Er bietet eine Vielzahl aktueller Themen aus den Bereichen Software-Reengineering und Software-Evolution, die gleichermaßen wissenschaftlichen wie

praktischen Informationsbedarf abdecken. In diesem Jahr gab es wieder Beiträge zu einem breiten Spektrum von Software-Reengineering- und Software-Evolutionsthemen.

In diesem Jahr fand der WSRE zum zwanzigsten Mal statt. Anlässlich dieses Jubiläums wurde der Workshop in besonderer Weise gestaltet. Vor allem der zweite der drei Workshop-Tage wurde als besonderer Jubiläumstag begangen. Nach einem Grußwort der WSRE-Initiatoren Jürgen Ebert und Franz Lehner haben mehrere Referenten, die in den vergangenen beiden Jahrzehnten den WSRE wesentlich mitgeprägt haben, in speziellen Retrospektiven-Vorträgen Rückblicke darauf gegeben, was im Bereich des Software-Reengineerings bereits erreicht wurde und auf welche besonderen Herausforderungen man dabei gestoßen ist. Im Einzelnen gab es Retrospektiven-Vorträge von Jochen Quante, Jens Borchers, Stefan Sauer, Rainer Koschke, Werner Teppe, Andreas Winter, Volker Riediger und Harry Sneed. Des Weiteren haben einige Anbieter von Reengineering-Werkzeugen und -Dienstleistungen ihre Werkzeuge in Kurzvorträgen und interaktiven Tool-Demos vorgestellt.

Die Organisatoren danken allen Beitragenden für ihr Engagement – insbesondere den Vortragenden, Autorinnen und Autoren. Unser Dank gilt auch den Mitarbeiterinnen und Mitarbeitern des Physikzentrums Bad Honnef, die es wie immer verstanden haben, ein angenehmes und problemloses Umfeld für den Workshop zu schaffen.

Für die FG SRE:

Jens Knodel, Caruso GmbH  
Torsten Görg, itemis AG  
Jens Borchers, Sopra Steria AG  
Matthias Gutheil, itemis AG

Für den L2S2:

Stefan Sauer, Universität Paderborn  
Robert Heinrich, KIT Karlsruhe  
Marco Konersmann, Universität Duisburg-Essen



er Technologie gibt. Die neue Technik muss vielmehr in zusätzlichem Nutzen stiften bzw. in aktuellen Anforderungen, in dem das System gestützt werden, besser (effizienter, effektiver) erfüllen als in der derzeitigen Technik.

Bislang auf dem Bestehen in den Informationssystemen lässt sich schlussfolgern, dass alle Komponenten (Mensch, Aufgabe, Technik und Organisation) Informationsträger und damit grundsätzlich im Alter unterliegen können. Die materiellen Systembestandteile sind zusätzlich der Verschleiß, Verfall zu berücksichtigen. Die Effektivität können die Materialgleichzeitigkeit auftragen und sich überlagern.

## 4 Bewertung

Nach unserer Auffassung ist in dem System in sozial-technischen Systemen, als bereits in der Existenz und währenddessen nicht vollumfänglich auf der derzeitigen tabellarischen Standards konstituierten in der Technologie aktualisiert wurden. Es existieren in der Technik mit den in der gestützten betrieblichen Aufgaben besser (effizienter, effektiver) erfüllt werden kann. Das biologische Alter der betrieblichen Menschen (nicht jedoch der Wissensstände) und der Verschleiß von Hardware ist für den Besten irrelevant. Daraus lassen sich - mit Hinblick auf die Inzidenz angeführt den Besten Definitionen - folgende Aussagen ableiten:

Das sogenannte Merkmal „schwerer rückwärts“ stellt in der Hinsicht der Qualitätsmerkmal Wartbarkeit und Weiterentwicklung dar. Die der Hinsicht der Kosten für die Durchführung der Anpassung auf die neue Technologie und geändert betrieblichen Anforderungen. Werden in der betrieblichen Praxis für die Wartungsaufgaben nicht ausreichende Ressourcen zur Verfügung gestellt, verleiht das Informationssystem. Mangelt es an Wartbarkeit bzw. Weiterentwicklung können es als Risikofaktor für die Ausbildung oder als Indikator für die Existenz in dem System hinsichtlich der Wartbarkeit bzw. Weiterentwicklung sein jedoch kein notwendiges oder hinreichendes Bedingung für die Existenz in dem System. Es ist weiterhin davon auszugehen, dass die Übernahme des Systems bzw. in Personalwechsel mit Wissensverlust einhergeht, was sich wiederum negativ auf die Weiterentwicklung auswirken kann. Demnach ist das Inzidenz genannt „übernehmen“ oder „rückwärts“ von Systemen als in Symptom für die Existenz oder Risikofaktor für die Ausbildung in dem System zu verstehen.

Das in der in der Definitionen genanntes Merkmal „geschäftskritisch“ ist kein Eigenschaft von Systemen, das nicht durch die Alterung abgeleitet ist. Es werden die rechtlich nicht geschäftskritischen Informationssysteme ausgeschlossen. Auf Grund unserer praktischen Erfahrung wissen wir, dass IT-Systeme existieren, die nach Jahren genutzt werden

Das in der „weiterentwickelt“ werden und danach problematisch abgeschaltet werden können. Dennoch ist auch für solche Systeme in der Verwertung der Besten des Systems möglich.

## 5 Zusammenfassung

Das Alter wurde auf dem hohen Abstraktionsniveau als Grad der Abweichung der im System verwerteten Technik gegenüber dem aktuellen Standard konstituierten in der Technologie unter Berücksichtigung der gestützten Aufgaben erklärt, wobei die Lebensdauer der betrieblichen Mitarbeiter und der Verschleiß der Hardware für den Besten des Systems irrelevant sind. Es wurde argumentiert, dass alle Komponenten in den Informationssystemen alter können und das mangelfähige Wartbarkeit, Weiterentwicklung und in dynamisch konstituierten Technologie als Risikofaktoren für die Ausbildung bzw. als Indikatoren für die Existenz in dem System hinsichtlich der Wartbarkeit bzw. Weiterentwicklung sind. Das Attribut „geschäftskritisch“ hat keinen Bezug zum Alter und ist daher kein Eigenschaft von Systemen.

## Literatur

- [1] G. Numann H.R. Hanssen. *Wirtschaftsinformatik I - Grundlagen betrieblicher Informationsverarbeitung*. Lucius & Lucius Stuttgart, 2002.
- [2] Prof. Dr. Wilhelm Müller & Prof. Dr. Dieter Abts. *Grundkurs Wirtschaftsinformatik - Eine kompakte und praxisorientierte Einführung*. Springer Vieweg, 2013.
- [3] René Rühl Lutz J. Hinrich, Armin Hinzel. *Wirtschaftsinformatik - Einführung und Grundlegung*. Springer, 2011.
- [4] Prof. Dr. Balzert. *Lehrbuch der Softwaretechnik: Entwurf, Implementierung, Installation und Betrieb*. Heidelberg: Spektrum Akademischer Verlag, 2011.
- [5] H. Snodgrass. *Softwarewartung und -wiederverwendung Bd I, Softwarewartung*. Rudolf Müller GmbH, 1991.
- [6] N. Oscar S. Dmyr, D. Stéphan. *Object-Oriented Reengineering Patterns*. Squared Brackets Associates, Switzerland, 2013.
- [7] D. Masak. *Legacy Software - Das lange Leben der Altsysteme*. Berlin Heidelberg: Springer Verlag, 2006.
- [8] Prof. Dr. Günther Ropohl. *Allgemeine Technologie, Eine Systemtheorie der Technik*. Universitätsverlag Karlsruhe, 2009.

# A Brief Survey of Object-Oriented Ideas

Johannes C. Hofmeister  
University of Passau

Janet Siegmund  
University of Passau

Sven Apel  
University of Passau

**Abstract:** Object-oriented programming is a widely known paradigm, supported in many modern programming languages, and is commonly associated with maintainable and understandable programs. To understand what makes up this paradigm and its effect on maintenance and understandability, we conducted a literature survey. Surprisingly, we found that object-oriented programming encompasses diverse sets of features and that there is no consensus on what features are necessary to unambiguously define object-oriented programming. We show that it is difficult to define object-oriented programming, but suggest that this lack of a consensual definition might actually have been the reason for the success of object-oriented programming in modern programming languages.

## 1 Introduction

One explanation for the success of the object-oriented paradigm lies on its positive effect on maintainability and understandability of source code. However, this relationship is not easily understood, because object-oriented programming looks different in every programming language, both syntactically and semantically. For example, classes are not essential to define object-oriented programming (e.g., ECMAScript < v6 uses objects, but not classes). More differences can be found between programming languages available today, and there appears to be no general consensus on how to define the object-oriented paradigm. This is problematic for researchers, practitioners and educators as it prevents them from evaluating the effects of this paradigm on maintainability and understandability.

To assess how far the different perspectives on object-oriented programming diverge, we conducted a literature scoping review. We sought to answer the following question:

*What are the differences and similarities between authoritative definitions of the object-oriented paradigm?*

## 2 Method

We identified related publications from the databases of ACM and IEEE, such as books, journal articles, and conference submissions, as well as some gray literature. We included publications including the keywords *object*, and *oriented* in their titles and snowballed into relevant cited publications. We applied an open card sorting strategy to score for relevance.

## 3 Results

The term *object-oriented* was coined by Alan Kay, who integrated ideas from Simula [3] into Smalltalk. When Kay was informally asked to provide a canonical definition of object-oriented programming, he answered: “*OOP to me means only messaging, local retention and protection and hiding of state-process, and extreme late-binding of all things.*” [9]. This forms a stark contrast to the definition of Wegner [15], who defines object-oriented programming languages as all languages that support *objects* and provide *classes* and *inheritance*. This chasm is further substantiated by other authors as summarized in Table 1. *Classes* are used to build objects, *uniformity* means “Everything is an object”, *inheritance* arranges classes into parent-child hierarchies, *messaging* is a metaphor for interaction between objects, *state* is objects’ capability to hold on to computed results, *encapsulation* refers to the idea that access to state should be isolated, and (subtype) *polymorphism* allows for objects substitutions.

There is more to each concept, and the authors’ valuations of these features are more subtle in the original papers; for brevity we classify them as essential (+), or non-essential (-) to define object-oriented programming.

Table 1: Essential and non-essential object-oriented features by author

	+	-
Classes	Wegner [15]	West [16]
Uniformity	Kay; Rentsch [9, 10]	Stroustrup [13]
Inheritance	Wegner [15]	Gamma [4]
Messaging	Kay [9]	Wegner [15]
State	Wegner [15]	Cook [2]
Encapsulation	Rentsch [10]	Stefik [12]
Polymorphism	Thomas [14]	Pierce [8]

In sum, the views differ and there is no agreement on what object-oriented programming *is*. However, its users seem to share a common goal: To create reusable, and maintainable software. The different technical features are believed to support the reduction of complexity, but there is disagreement which features support this overarching goal. For example, Thomas [14] outlines that *inheritance* means to reuse code of existing classes, which improves maintainability, in that there is less code to maintain. However, Gamma et al. [4] contest this idea and outline that inheritance breaks encapsulation and reduces the maintainability of software systems. They suggest to favor *composition* over inheritance. In sum, it remains difficult to evaluate the impact of different features on

the maintainability of software systems.

## 4 Discussion

We outlined several features that are in the focus of ongoing discussions and conclude that internally, object-oriented programming is not a uniform concept, but a set of different features, ideas and techniques that are easier to characterize than to define.

While the mechanisms used are negotiable, they are embraced by the goal to create maintainable software. In the object-oriented view, maintainability is supported by different mechanisms of reuse (e.g., polymorphism, composition), but also mechanisms of decoupling (e.g., encapsulation, messaging, uniformity), in that aspects of the program are split into different modules. The driver behind these concepts is *comprehensibility*. Software is created by humans who model and encode aspects of a problem domain into computer programs. The mechanisms of comprehension have been discussed in literature: Object encodings match *problem decompositions*, and encapsulation is a means of *information hiding* [7]. Compositional structures and the idea of encapsulation allow programmers to create *abstractions*, and add meaning to programs [11]. Only when programmers comprehend a program are they capable to alter its functioning. If they do not understand the program, they cannot assess the effects of their alterations. Originally, objects were invented to support program comprehension [3], but today it is poorly understood if or how objects actually facilitate comprehension processes.

However, Cognitive Psychology offers some explanations: Objects are special in their capability to stand in for *entities* from the real world. Aldrich [1] calls them *service abstractions* and Kay outlines [5] that objects may stand in for higher level behavioral goals. Snyder [11] states that "objects provide services" and "all objects embody abstractions". These views indicate that objects are capable of representing abstract things, such as behaviors, goals, or services, and are not limited to representing physical things. Thus, programmatic objects work as *symbolic* encodings, in that they pose as representations of *categories*, or, in other words, objects are encodings of *concepts*.

Categories are collections of items in the world that we choose to treat as equivalent for some purpose, and concepts are representations of these categories. Psychological research shows that people employ various strategies to form categories, for example, feature based (e.g., Birds: eagle, sparrow, penguin), goal derived (e.g., Picnic food), relational (e.g., Competitions: war, chess, F1), role governed (e.g., Barriers: membrane, roadblock, locked door), or thematic (e.g., Kitchen items: stove, chef, pan) [6]. Objects are capable of mimicking each of these strategies (e.g., inheritance mediates relation, classes can be used to categorize objects, and subtype polymorphism allows the reuse of objects in different thematic or goal derived

contexts). In a nutshell, objects appear to be natural but not in that they resemble physical entities, but because they offer flexibility to encode a person's conceptual understanding, and thus model and simulate problems from the real world.

This might be one reason why object-oriented programs are linked with high maintainability and understandability: Since developers can map programmatic objects to real-world objects, they can integrate the properties and behavior of programmatic objects to their existing knowledge of real-world objects, thereby using according skills, for example, to manipulate them or categorize them.

To better understand the role of object orientation for maintainability and understandability, we will (1) extend the literature review by additional papers and an automated approach to extract features of object-oriented programming, and (2) conduct studies with developers to evaluate whether their mental representation of real-world objects and programmatic objects is actually comparable.

**Acknowledgements:** Work supported by DFG grant SI 2045/2-1.

## References

- [1] J. Aldrich. The power of interoperability: Why objects are inevitable. In *SPLASH '13, Indianapolis, IN, USA, October 26-31, 2013*, pages 101–116. ACM Press, 2013.
- [2] W. R. Cook. On understanding data abstraction, revisited. *ACM SIGPLAN Notices*, 44(10):557–572, 2009.
- [3] O.-J. Dahl, B. Myrhaug, and K. Nygaard. *SIMULA 67 Common Base Language*, (Norwegian Computing Center. Publication). 1968.
- [4] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.
- [5] A. C. Kay. The early history of smalltalk. *SIGPLAN Not.*, 28(3):69–95, 1993.
- [6] A. B. Markman and J. R. Rein. The nature of mental concepts. In D. Reisberg, editor, *The Oxford Handbook of Cognitive Psychology*, chapter 21, pages 321–329. Oxford University Press, Oxford, 2013.
- [7] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, 1972.
- [8] B. C. Pierce. *Types and Programming Languages*. The MIT Press, 2002.
- [9] S. Ram. Dr. alan kay on the meaning of "object-oriented programming", 2003.
- [10] T. Rentsch. Object oriented programming. *SIGPLAN Not.*, 17(9):51–57, 1982.
- [11] A. Snyder. The essence of objects: Common concepts and terminology. *IEEE Software*, 1993.
- [12] M. Stefik and D. G. Bobrow. Object-oriented programming: Themes and variations. *AI magazine*, 6(4):40, 1985.
- [13] B. Stroustrup. What is object-oriented programming? *IEEE Software*, 5(3):10–20, 1988.
- [14] D. A. Thomas. What's in an object? *BYTE Magazine*, 1(5):231–240, 1989.
- [15] P. Wegner. Dimensions of object-based language design. In *OOPSLA '87, Orlando, Florida, USA, October 4-8, 1987, Proceedings.*, pages 168–182, 1987.
- [16] D. West. *Object Thinking*. Microsoft Press, Redmond, WA, USA, 2004.