Issues in Agent-Oriented Software Engineering

Jürgen Lind

German Research Center for AI (DFKI)
Im Stadtwald B36
D-66123 Saarbrücken, Germany
lind@dfki.de

Abstract. In this paper, I will discuss the conceptual foundation of agent-oriented software development by relating the fundamental elements of the agent-oriented view to those of other, well established programming paradigms, especially the object-oriented approach. Furthermore, I will motivate the concept of autonomy as the basic property of the agent-oriented school and discuss the development history of programming paradigms that lead to this perspective on software systems. The paper will be concluded by an outlook on how the new paradigm can change the way we think about software systems.

1 Introduction

Agents and multi-agent systems are currently one of the most interesting research fields in the computer science community; especially the natural way of capturing the structure and the behavior of complex systems has stimulated this huge interest. But is this enough to make agent-oriented software engineering (AOSE) a new software paradigm? What makes the idea distinctive from other approaches? How does it fit in a more general picture of software engineering?

In this paper, I will present my personal viewpoint on agent-oriented software engineering firstly by discussing the interrelationships of AOSE concepts (agent, agent architecture, role, etc.) and secondly by relating AOSE to other programming paradigms. Especially the relation between object-oriented and agent-oriented methods is particularly interesting because they seem to be closely related. In order to clarify their relationship, I will describe the levels of abstraction that are involved in a certain programming paradigm in general and of object-orientation and agent-orientation in particular. I will then identify aspects they have in common as well as their main differences. Furthermore, I will point out what could be the major contributions of the agent oriented paradigm to software engineering and provide an outlook on how the new paradigm can change the way we think about software systems.

2 Aspects of programming paradigms

The term "programming paradigm" is extremely fuzzy because it is often used to capture a set of different software-related aspects under a particular catch-phrase. These

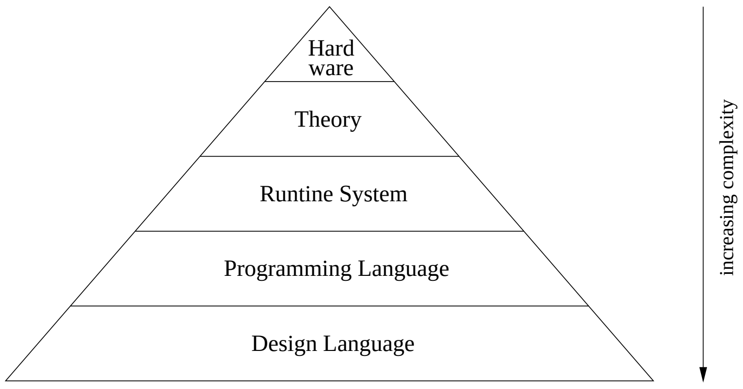

Fig. 1. Levels of Abstraction

different aspects are often located on different levels of abstraction and their interrelationships are seldom explicitly formulated. In this paper, I will use the triangle shown in Figure 1 to describe the different levels of abstraction that in my view make up a programming paradigm. The form a triangle was chosen to express the fact that the number of concepts (and therewith the complexity) on a particular level of abstraction increases on higher levels. Furthermore, a layered approach is quite common in computer science theories to clearly separate the concepts on different levels of abstraction. The main advantage of a layered approach is that no knowledge of lower levels is necessary to understand and to work with higher level concepts because ideally, each level of abstraction represents a conceptually closed framework. In reality, unfortunately, the higher level theories are not only much more complex then lower level ones, but they are often incomplete [22]. Therefore, it is often necessary to combine several higher level theories to obtain a full coverage of the part of the world that should be modeled.

Note furthermore, that the distinctions between the different levels are not too sharp. Because of the fact that most programming models are assumed to be essentially equal in their computational power (Church’s thesis), any programming model can be implemented in terms of any other model. Thus, it is possible to write object-oriented software in a purely imperative programming language or to implement a deductive database in an object-oriented framework. In the following sections, I have therefore tried to produce a break-down of concepts that clearly separates intra-model aspects and that allows for an inter-model comparison of these concepts. I am well aware that some concepts can be shifted along the abstraction hierarchy, but I think that the current assignment to a particular level is adequate.

2.1 Hardware

The first level of abstraction encapsulates the architecture that is implemented in the computer hardware. Today, most computers still have the von Neumann architecture that was introduced in the late 1940s [12]. The architecture consists of a *processor* that is subdivided into units for computation and control and a *memory* store that holds the instructions and the data of the program.

This architecture is still common in modern computers although it has been greatly optimized by using techniques such as pipelining, caching or parallelism to speed up computation. A recent trend in the hardware community is to turn away from integrated, large-scale systems and towards networks of normal personal computers that jointly work on a computationally demanding task. These virtual supercomputers combine the advantage of lower costs through the use of standard hardware with an extreme scalability that allows to add more computational resources whenever this is necessary. In one vision on the future of the Internet [23], the entire net becomes a virtual supercomputer that makes individual computational power obsolete.

However, whether sequential, parallel or distributed, from the point of view of a programming paradigm, all hardware looks the same. There have been attempts to build hardware architectures that implement a particular programming paradigm directly into the hardware device, but none of these attempts has been successful. Therefore, we can safely assume that all programming paradigms share the same ground.

2.2 Theories

On the next higher level of abstraction, however, things are different. Theories are conceptualizations of a particular computational model that abstracts away from the characteristics of the hardware. The first theories were aimed at capturing the in-principle capability of a computational device in order to allow for general statements about what can be automatically computed and what cannot [41]. Turing's theory, for example, is a radical mathematical conceptualization of the von Neumann architecture that enables us to formally analyze all possible programs that can be executed on such an architecture. Other computational theories are intended as tools to help the programmer to express the ideas of what a program is supposed to do more naturally. An early computational theory that was meant as the foundation of a "natural" way of programming is declarative programming [17] but it has been demonstrated by empirical investigations in cognitive psychology that this claim does not necessarily hold true [32].

Let's start the comparison of the object-oriented and agent-oriented issues with the entities that are handled on this level of abstraction. In the object-oriented world, these entities are the *objects*. An object can be anything ranging from a concrete entity from the real world to a conceptual entity that only exists in the designers head. Each object within the system is associated with a particular *class* that determines the objects basic properties. Classes can be linked with each other in several ways. Probably the best known relation between two classes is *inheritance* that models a conceptual extension of a common base specification. During their lifetime, objects communicate by sending *messages* to each other. These messages can be used to request services from the receiving object such as to provide internal information or to change the current state. Although there are several additional concepts in the object-oriented paradigm I will restrict myself to this brief introduction and refer the reader unfamiliar with object-oriented concepts to the available literature, eg. [4]. In summary, the collection of object-oriented concepts is clear and manageable in size and does not vary greatly in different object-oriented approaches.

In the agent-oriented universe, on the other hand, we are faced with the first serious problem as there is no single agreed definition of the entities that are dealt with. The existing *agent theories* are more or less built upon one out of two widely accepted notions of agency [44]. In the *strong notion* of agency, an agent is modeled in terms of mentalistic notions such as beliefs, desires and intentions. Furthermore, the strong notion requires that these mental concepts have an explicit representation within the implementation of the agent. Thus, this notion forces a *white-box* on the agent. The *weak notion* of agency, on the other hand, requires only a *black-box* view on the agent in that it defines an agent only in terms of its observable properties. According to this definition, an agent is anything that exhibits autonomy, reactivity, pro-activity, social ability [44].

In my opinion, these two notions of agency are both too strict. I would argue for a more pragmatic definition of agency that allows the designer to decide what should be an agent regardless of a particular implementation or a minimal degree of external properties. I call this the *very weak notion* of agency. To explain why this absence of formal aspects still makes sense, I have to fall back upon a famous article from the early days of Artificial Intelligence.

In [22], the author argues that it is useful to ascribe mental qualities such as beliefs, goals, desires, wishes etc. to machines (or computer programs) whenever it helps us to understand the structure of a machine or a program or to explain or predict the behavior of the machine or the program. McCarthy does not impose any constraints such as a minimal required complexity onto the entities that we want to ascribe mental categories or onto the mental categories that we would like to use. In his view, ascribing mental qualities is a means of understanding and of communication between humans, ie. it is a purely conceptual tool that serves the purpose of expressing existing knowledge about a particular program or its current state.

> "All the [. . .] reasons for ascribing belief's are epistemological; i.e. ascribing beliefs is needed to adapt to limitations on our ability to acquire knowledge, use it for prediction, and establish generalizations in terms of the elementary structure of the program. Perhaps this is the general reason for ascribing higher levels of organization to systems."

To illustrate why this point of view is reasonable, McCarthy uses the example of a program that is given in source code form. It is possible to completely determine the programs behavior by simulating the given code, ie. no mental categories are necessary to describe this behavior. Why would we still want to use mental categories to talk and reason about the program? In the original paper, McCarthy discusses several reasons for this. In the following list, I have selected those reasons that seem to be most relevant to me:

1. The programs state at a particular point in time is usually not directly observable. Therefore, the observable information is better expressed in mental categories.
2. A complete simulation may be too slow, but a prediction about the behavior on the basis of the ascribed mental qualities may be feasible.
3. Ascribing mental qualities can lead to more general hypothesis about the programs behavior then a finite number of simulations.

4. The mental categories (eg. goals) that are ascribed are likely to correspond to the programmers intentions when designing the program. Thus, the program can be understood and changed more easily.
5. The structure of the program is more easily accessible then in the source code form.

Especially the fourth point in the above enumeration is extremely important for AOSE because the task of understanding existing software becomes increasingly important in the software industry and is likely to outrange the development of new software [1]. Thus, if it becomes easier to access the original developers idea (that is eventually manifested in the design) it becomes easier to understand the design and this leads to higher cost efficiency in software maintenance.

A more general conclusion from McCarthy's approach is the idea that *anything can be an agent*. This view has been discussed from controversial points of view [44] and it has been argued that it does not buy us anything whenever the system is so simple that it can be perfectly understood. I do not agree with this. In my view, the conceptual integrity that is achieved by viewing every intentional entity – be it a simple as it may – in the system as an agent leads to a much clearer system design and it circumvents the problem to decide whether a particular entity is an agent or not. In my personal experience, this problem can be quite annoying during the design phase whenever two software designers have different views.

In the above paragraphs, I have identified the basic structural elements of object-orientation and agent-orientation, respectively. Now I will outline some of the basic concepts of describing and arranging these elements and point out some fundamental similarities that can be identified.

As I have already said above, the basic descriptional element is object-oriented programming is the class. A class definition specifies the class variables of an object and the methods the object accepts. Classes can be linked with each other via several forms: one class inherit from another class such that the new class is an extension of the existing class, instances of two classes can collaborate with each other by exchanging messages, and finally they can have a structural connection in that one instance of a class contains an instance of the class.

These concepts correspond to the agent-oriented world by replacing class with *role*, state variable with *belief/knowledge* and method with *message*. Thus a role definition describes the agent's capabilities, the data that is needed to produce the desired results and the requests that trigger a particular service. Besides this fundamental relation, there are many other conceptual similarities between object-orientation and agent-orientation that can be mapped onto each other. Due to the limited space, however, these are briefly summarized in Table 1.

Turning away from the conceptual issues and similarities of the two programming approaches, we will now come to more technical aspects of the runtime environment and discuss the general structure for object-oriented and agent-oriented systems, respectively.

2.3 Runtime System

The runtime system of a particular programming paradigm provides the environment for the program interpretation and these environments can be radically different. In

	OOP	AOP
Structural Elements		
	abstract class	generic role
	class	domain specific role
	class variables	knowledge, belief
	methods	capabilities
Relations		
	collaboration (`uses`)	negotiation
	composition (`has`)	holonic agents
	inheritance (`is`)	role multiplicity
	instantiation	domain-specific role + individual knowledge
	polymorphism	service matchmaking

Table 1. Mapping OOP to AOP

the more simple forms, they are restricted to administrative tasks such as managing the heap or they provide slightly more elaborate services such as garbage collection. However, there also exist very complex runtime environment that provide complete reasoning engines for logic programming [17] that are for example used in declarative programming languages such as Prolog [5].

Objects and agents and the various relationships that exist between them within their respective programming model are conceptual abstractions that require an implementation such that they can be used by higher levels of abstraction. In the following paragraphs, I will divide the implementation of the theoretical concepts into the implementation of the entities themselves and an implementation of a meta-level that manipulates the basic entities.

In an object-oriented runtime system, the objects are statically represented by the *object architecture*. This architecture is usually quite simple as it only contains the current state of the object and the relation to the objects class (which determines the operations that can be performed on the object). An object is usually represented as arbitrary collection of data elements with associated functions and the granularity of objects is potentially not limited. However, efficiency issues dictate that not every entity is modeled as an object and so in reality this conceptual benefit is slightly weakened. The *object management system* is responsible for representing the relations such as inheritance between the defined classes and object manipulation such as creating or destroying objects. Furthermore, the object management system is also responsible for dynamic aspects such as method selection of polymorphous objects, exception handling or garbage collection.

In an agent-oriented runtime system, things are distinctly more complicated although similar in their general structure. The basic entities are the agents that are implemented by their *agent architecture*. Agent architectures are often built upon a particular theory such as BDI [35] and establish the link between the abstract concepts “agent” and “role” in that they provide the runtime environment for the role descriptions that make up the agent. Thus, we have the fundamental relation [21]

$$\textit{agent} = \textit{roles} + \textit{architecture}$$

However, agent architectures are far more complex then the object architecture, especially because of the dynamic aspects that must be dealt with. Because of the richness of the agent-oriented world, there exists a large number of different agent architectures [27, 28, 15]. Due to the vast number of approaches, it is impossible to identify *the* best or most general architecture. However, the smallest common denominator seems to be the basic *perceive* – *reason* – *act* cycle that is oriented at the minimal agent model of [36]: in each iteration, the agent perceives the state of its environment, integrates the perception in its knowledge base that is used to derive the next action which is then executed. This generic cycle is a useful abstraction as it provides a black-box view on the agent architecture and encapsulates specific aspects.

The task of the *agent management system* as the meta-level of an agent based runtime environment is to provide a “life-space” for the agents, ie. a collection of mechanisms that enables the agents to get in contact with each other. To enable agents of different designers to interact with each other, it is necessary to standardize the basic services that are provided by agent management system. One such standard is defined in [10].

2.4 Programming Language

On this level of abstraction, the syntactical framework for the manipulation of the entities on the runtime level is defined. The programs that are written in a particular programming language are either directly interpreted by the runtime system or they are compiled into an intermediate format that is understood by the runtime system or directly to assembler code.

The syntactical constructs that are provided by the programming language should allow the programmer to use the underlying semantic concepts efficiently and to express the intended functionality of the program elegantly. For example, it is generally possible to implement a particular conceptual model with any general purpose language, e.g. it is possible to write object-oriented programs in C, but in general, it is much easier and more comfortable for the programmer if the terms of the conceptual framework can be used directly. Even an integration of several conceptual models into a single high level programming language can be problematic as is often difficult to find a good combination of concepts that is not overwhelming for the average user and then to find a concise syntactical representation for these different concepts.

I think that object-orientation as well as agent-orientation are such general concepts that can be attached to almost any other programming language. In the case of object orientation, this approached work for languages such C, leading to C++ [38], Cobol (ObjectCobol [9]), perl [42] and numerous other languages. But not only imperative languages have been enhanced with objects. The Mozart programming system [34], for example, provides a very elegant combination of constraint-logic programming with object-oriented concepts.

In the context of agent-oriented software engineering, these trends are not so clear until now. Currently, there is no – at least to my knowledge – widely accepted agent-oriented programming language that goes beyond the experimental state. However, some approaches are designed as an extension of established languages, eg. JAM Agents [14] that combine agent-oriented concepts with Java [39].

2.5 Design Language

Design languages are further abstractions from a particular programming language that aim at the conceptual modeling of a system at a more coarse grained level. Design languages often use graphical notations that make it easier fro the designer to access the overall system structure. Probably the currently best known design language is the *Unified Modeling Language (UML)* [3] that tries to integrate several, until then separated design notations, under a common hat. The UML provides a variety of structural elements with well defined semantics that can be flexibly combined into diagrams that capture different aspects of a software system. The core UML language can thus be used to describe a software system from the requirements specification to the final design. An example for using the UML within the context of agent-based systems is discussed in [7]. Due to the general nature of the core UML, however, it is not always suited for all problem areas, and therefore, extensions that cover special aspect have already been proposed [11]. One way is to extend the UML by providing new structural elements and diagrams that enhance the expressive power of the base language. This way is favored by the OMG/FIPA in the development of AGENTUML [31] which proposes an extension of the UML with respect to agent-oriented concepts. As part of the AGENTUML in the FIPA standard [2], [30] suggests an extension of the UML by a completely new diagram type called *protocol diagrams*. These diagrams combine elements of UML interaction diagrams and state diagrams to model the roles that can be played by an agent in the course of interacting with other agents. The new diagram type allows for the specification of multiple threads within an interaction protocol and supports protocol nesting and protocol templates based on generic protocol descriptions.

In a more general sense, however, design languages should not necessarily be constraint to modeling aspects of the system. In my personal view, I would count general software architecture frameworks or frameworks for a particular application area to design languages as well. The reason for this view is that these frameworks provide their own set of structural abstractions that represent a “language” on this particular level of abstraction.

In the object-oriented community, examples for such frameworks include Java Beans [40] as a means to provide off-the-shelf components together with flexible interconnection mechanisms between the basic structural elements, or software development environments such as Visual C++ [25] that focus on a support for the development of graphical user interfaces. In the latter case, the structural elements of the design language are graphical elements that are combined according to a given grammar that regulates how different elements can be put together.

In the agent-based world – although a relatively new area –, a large number of different frameworks already exists. This may be due to the fact, that the increasing

	Machine Language	Structured Programming	Object-Oriented Programming	Agent-Oriented Programming
Structural Unit	Program	Subroutine	Object	Agent
Relation to Previous level		Bounded unit of Program	Subroutine + persistent local state	Object + independent thread of execution + Initiative

Table 2. Historic development of programming paradigms [33]

complexity can only be dealt with by using adequate tool support. Examples for agent-based design languages range from source-level frameworks such as SIF [37] up to complex and powerful tools such as the ZEUS toolkit [29] from British Telecom that provides drag-and-drop mechanisms for putting together multi-agent applications.

3 A Historic Perspective

In this section, I will discuss a few historic aspects in the development of programming paradigms that can be helpful in understanding why the agent-oriented approach is a natural successor to the prior development.

In [33], Table 3 is used to capture the historic development from machine language to agent-oriented programming. In the early days of programming, a program was thus seem as a monolithic block without any inherent structure. This view was subsequently changed in that it was recognized that a program is made up from several smaller structural units, ie. subroutines. However, the concept of subroutines alone was not powerful enough as it emphasized the control flow aspect of programming and neglects the data that is involved. Consequently, the view changed a second time, this time grouping data and computation together in a single structural unit called an “object”. Currently, we are faced with the third change of perspective, leading away from merely passive objects and facing towards active structural units which we call “agents”.

I like the above presentation of the historic development because I think that it captures the main ideas in a concise form. However, I am not completely satisfied with the characterization of agents in the above table. While the requirement of an independent thread of execution sounds very technical, the term “initiative” is to fuzzy to be operationalized. To draw on the basic ideas of [33] but to develop a more coherent structure, I suggest the three-step characterization shown in Figure 2.

In the first step, programs are seen as a collection of *functions* that establish a well-defined goal. These functions can be described as an imperative sequence of statements (*imperative programming*), as a collection of mathematical expressions that are linked together (*functional programming*) or as a set of goals without imposing a particular way of achieving the goal onto the interpreter (*declarative programming*).

In the next development step, a program is interpreted in terms of the *data* that is manipulated and the *functions* that operate on that data. This leads to structured programming where semantically related aspects of the program are spatially related. An

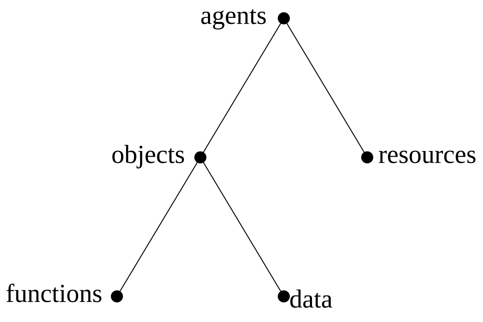

Fig. 2. Historic development

even stronger and explicit relation between data and functions is introduced by abstract data types, eventually leading to object-oriented programming.

In the final step of the characterization, the objects are augmented with *resources* such as computation time, that can be freely used. This freedom in the (internal) resource allocation process lead to the concept that I find most fundamental for agent-oriented programming: *autonomy*. Although the weak notion of agency has identified autonomy as a central concept of the agent-oriented viewpoint, it was only credited as one among others. I would argue, on the other hand, that autonomy is more fundamental then the other aspects of the the weak notion and that it is even a prerequisite for the others. For example, pro-activeness can only be achieved when the agent is free to decide when to become active; the same argument holds for reactivity.

The idea of agents as autonomous agents is so striking and revolutionary because it leads to a new way of thinking about software systems. Such a system is no longer a collection of passive objects. Rather, these objects have a “life of their own”, ie. they are perceived and modeled by the designer as active entities. This view on complex systems is completely different from traditional approaches in that it explicitly accepts the fact the system designer is not responsible for specifying the systems dynamics down to the least bit. Instead, the designer sets out the initial state and specifies the initial goals of the autonomous agents and then the system takes over. In such a system, there is no such thing as the “central scrutinizer” [46] that controls everything. Rather, the ongoing interactions determine the overall system behavior [13].

Another major advantage of the agent-oriented view is that it supports the principle of locality even better then the object-oriented view does. In object-oriented systems, the control-flow specification is spread all over the entire program code. The agent-oriented view introduces a further tool for conceptual grouping that comes with the agents well defined bounds [19]. All elements that make up the control-flow of a particular agent are grouped under the common concept, making it easier to identify larger units of the program that belong together semantically.

4 The bottom line

After the sobering remarks about the basic similarities of the agent- and object-oriented approaches one may be tempted to conclude that agent-orientation are just the emperor's new clothes. But that is not what I was trying to say. Even if the technical contributions or agent-oriented software engineering are not really revolutionary the conceptual contribution is nonetheless huge. Agent-oriented software engineering provides an epistemological framework for effective communication and reasoning about complex software system on the basis of mental qualities. It provides a consistent new set of terms and relations that adequately capture complex systems and that support easier and more natural development of these systems.

As an example for the importance of a clear terminological framework, consider abstract data types (ADTs) and objects. It is argued in [43], that objects are essentially the same thing then ADTs that were introduced years earlier. But: why do programmers prefer objects over ADTs? I think because the terminological framework provided by object-oriented approaches allows the programmer a more natural way of modeling because it allows for thinking in terms of the real world that should be modeled by a software system. Furthermore, I think that it will be a major reason for the success of the agent-oriented view that programmers already use some sort of mentalistic notion to develop their object-oriented systems that is subsequently translated into object-oriented terms. This additional transformation can be dropped as soon as the adequate tools for expressing the ideas directly in the already used terminology become available.

As a second point that I have explained above, I think that adding autonomy as an accepted property of formerly passive objects is the main contribution of the agent-oriented view. It leads to a completely different modeling approach that stimulates a system design built upon the desirable properties [6] of loose coupling between system components with a high cohesion of these components.

I shall now return to the initial question of the paper that was whether agent-oriented software engineering is really a new programming paradigm or not. To answer this question, consider the following quote from the Webster On-line Dictionary [24]

> Main Entry: par·a·digm
> Pronunciation: `'par-&-"dIm` also `-"dim`
> Function: *noun*
> Etymology: Late Latin *paradigma*, from Greek *paradeigma*, from *paradeiknynai* to show side by side
> Date: 15th century
> **1 : example, pattern**; *especially*: an outstandingly clear or typical example or archetype
> **2 :** an example of a conjugation or declension showing a word in all its inflectional forms
> **3 :** a philosophical and theoretical framework of a scientific school or discipline within which theories, laws, and generalizations and the experiments performed in support of them are formulated

According to this definition, the answer to the above question is clearly "yes" because agent-oriented software engineering provides us with the required new framework, built upon the basic property of autonomy, that allows for the modeling and understanding of agent-based applications. Furthermore, I think that the agent-oriented view is a necessary prerequisite for accepting artificial intelligence at all because I think that we must get used to ascribing basic qualities such as goal, beliefs, desires before we can ascribe "intelligence" to a machine.

5 Where next?

It must be the goal for the agent community to broaden the acceptance of the new paradigm among the people who really develop software, ie. software engineers. But just as it was the case with object-oriented technology, I do not believe that this acceptance will develop quickly. Object-oriented technology was around for about 10-15 years before it became a widely accepted and naturally used software engineering discipline. So the question one may ask in this respect is why it takes so long for a new paradigm to become state of the art? An interesting answer to this question is provided in Kuhn's theory about the *Structure of Scientific Revolutions* [18]. According to Kuhn's theory, scientific development is not a continuous flow, but rather a sequence of disjoint revolutions. Every such a revolution is preceded by a phase of normal scientific activities in which the researches use the current state of the art (the current paradigm) as the general background of their daily work and the research questions are draw from yet unsolved problems of the current paradigm and can in principle be solved within the existing framework. From time to time, however, a question is raised or a phenomenon is observed that cannot be answered or explained within the current paradigm. These anomalies require a radical change of perspective, ie. a new general research paradigm that can deal with the newly observed phenomena. This is then called a revolution. Ideally, the new paradigm should also capture the past experiences although this is not always possible. As an example for this sort of scientific development, consider Newton's theory on mechanics. Newton's mechanics was the research framework for several hundred years until several observations on the atomic level could not be explained in Newton's theory. This lead to the development of quantum mechanics that were able explain the observations on the atomic level.

The major point in Kuhn's theory is, that the new research paradigm is not introduced into the research by established researchers that "convert" to the new paradigm. Rather, it is introduced by the upcoming generation of young researchers that grow up in the spirit of the new paradigm and that they naturally accept as the general framework. Scientific history is full of examples for this process. The above mentioned theory of quantum mechanics is such an examples, as is Darwin's theory on the origin of species [8]. On a much more specific level, this observation is also true for object-oriented software development. While some established researches neglected the novelty in the concepts [43], it was readily accepted by the younger generation and it is now a widely accepted programming paradigm.

In the near future of agent-oriented software engineering, however, it is necessary to make the main contributions accessible to the people that should use it. Therefore, we need conceptual frameworks to such as described in [16, 20, 26, 45] that support the development of agent-oriented applications.

References

1. Helmut Balzert. *Lehrbuch der Software-Technik*, volume II. Spekrum Akademischer Verlag, 1998.
2. Bernhard Bauer, Jörg P. Müller, and James Odell. Agent UML: A Formalism for Specifying Multiagent Software Systems. In *Proceeedings of the First International Workshop on Agent-Oriented Software Engineering (AOSE-2000) held at the 22nd International Conference on Software Engineering*, Limerick, Ireland, 2000.
3. G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide*. Addison Wesley, 1999.
4. Grady Booch. *Object-Oriented Analysis and Design With Applications*. Addison-Wesley, 1994.
5. W. F. Clocksin and C. S. Mellish. *Programming in Prolog*. Springer Verlag, 1994.
6. S. D. Conte, H. E. Dunsmore, and V. Y. Chen. *Software Engineering Metrics and Models*. The Benjamin/Cummings Publishing Company, 1996.
7. Ralph Depke, Reiko Heckel, and Jochen Malte Küster. Requirement Specification and Design of Agent-Based Systems with Graph Transformation, Roles, and UML. In *Proceeedings of the First International Workshop on Agent-Oriented Software Engineering (AOSE-2000) held at the 22nd International Conference on Software Engineering*, Limerick, Ireland, 2000.
8. Adrian Desmond and James Moore. *Darwin*. Rowolt, 1994.
9. E. Reed Doke and Bill C. Hardgrave. *An Introduction to Object Cobol*. John Wiley & Sons, 1998.
10. FIPA. Fipa '98 specification parts 1–13, version 1.0, 1998. The Foundation for Intelligent Physical Agents.
11. R. France and B. Rumpe, editors. *UML99 - The Unified Modelling Language - Beyond The Standard*, number 1723 in LNCS. Springer, 1999.
12. J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., 1990.
13. Michael N. Huhns. Interaction-oriented programming. In *Proceeedings of the First International Workshop on Agent-Oriented Software Engineering (AOSE-2000) held at the 22nd International Conference on Software Engineering*, Limerick, Ireland, 2000.
14. Intelligent Reasoning Systems. Jam agent architecture, 2000. http://members.home.net/marcush/IRS/.
15. C. G. Jung. *Theory and Pratice of Hybrid Agents*. PhD thesis, Universität des Saarlandes, 1999.
16. Elizabeth A. Kendall. Agent software engineering with role modelling. In *Proceeedings of the First International Workshop on Agent-Oriented Software Engineering (AOSE-2000) held at the 22nd International Conference on Software Engineering*, Limerick, Ireland, 2000.
17. Robert Kowalski. *Logic for Problem Solving*. North Holland, Amsterdam, 1979.
18. Thomas S Kuhn. *The structure of scientific revolutions*. Univ. of Chicago Press, 2nd edition, 1975.
19. Susan E. Lander. Issues in Multiagent Design Systems. *IEEE Expert*, April 1997.
20. Jürgen Lind. The MASSIVE development method for multiagent systems. In *Proceedings of the Fifth International Conference on the Practical Application of Intelligent Agents and Multi-Agents*, Manchester, UK, 2000.
21. Jürgen Lind. MASSIVE*: Software Engineering for Multiagent Systems*. PhD thesis, University of the Saarland, 2000.
22. John McCarthy. Ascribing mental qualities to machines. In Martin Ringle, editor, *Philosophical Aspects in Artificial Intelligence*. Harvester Press, 1979.

23. Scott McNealy. Scott says... kick butt and have fun”. *Sun Microsystems*, 1996. http://www.sun.com/960601/cover/.
24. Merriam-Webster. Wwwebster dictionary, 2000. http://www.m-w.com.
25. Microsoft Corporation. Visual c++, 2000. http://msdn.microsoft.com/visualc/.
26. Simon Miles, Mike Joy, and Michael Luck. Designing agent-oriented systems by analysing agent interactions. In *Proceeedings of the First International Workshop on Agent-Oriented Software Engineering (AOSE-2000) held at the 22nd International Conference on Software Engineering*, Limerick, Ireland, 2000.
27. J. P. Müller. Control Architectures for Autonomous and Interactin Agents: A Survey. In L. Cavedon, Anand Rao, and Wayne Wobcke, editors, *Intelligent Agent Systems: Theoratical and Practical Issues*, number 1209 in LNAI, 1996.
28. Jörg P. Müller. The Right Agent (Architecture) to do the Right Thing. In *Intelligent Agents V — Proc. of the ATAL-98*, volume 1555 of *LNAI*, 1998.
29. Hyacinth S. Nwana, Divine T. Ndumu, Lyndon C. Lee, and Jaron C. Collins. ZEUS: A tool-kit for building distributed multi-agent systems. *Applied Artifical Intelligence Journal*, 13(1):129–186, 1999.
30. James Odell, H. Van Dyke Parunak, and Bernhard Bauer. Representing agent interaction protocols in uml. In *Proceeedings of the First International Workshop on Agent-Oriented Software Engineering (AOSE-2000) held at the 22nd International Conference on Software Engineering*, Limerick, Ireland, 2000.
31. OMG and FIPA. Agent working group. `http://www.objs.com/isig/wg-agents06-minutes.html`, 1999.
32. Tom Ormerod. Human cognition and programming. In *Psychology of Programming*. Academic Press Ltd., London, 1990.
33. H. V. Parunak. Blue-Collar Agents: Keynote of the PAAM99 conference. http://www.erim.org/∼van/Presentations, April 1999.
34. Programming Systems Lab. The mozart programming system. University of the Saarland, 1999. http://www.mozart-oz.org.
35. A. S. Rao and M. Georgeff. BDI Agents: from theory to practice. In *Proceedings of the First International Conference on Multi-Agent Systems (ICMAS-95)*, pages 312–319, San Francisco, CA, June 1995.
36. Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, 1995.
37. M. Schillo, J. Lind, P. Funk, C. Gerber, and C. Jung. SIF - The Social Interaction Framework System Description and User’s Guide to a Multi-Agent System Testbed. Technical Report TR-99-02, DFKI GmbH, 1999.
38. Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, Massachusetts, 1987.
39. Sun Microsystems. The Java Programming System, 1999. http://java.sun.com.
40. Sun Microsystems. Java Beans, 2000. http://java.sun.com/beans.
41. Alan M. Turing. On computable numbers, with an application to the entscheidungsproblem. *Proceedings of the London Mathematical Society*, 2(42), 1937.
42. Larry Wall, Randal L. Schwartz, and Tom Christiansen. *Programming Perl*. O’Reilly & Associates Inc., 2nd edition, 1996.
43. N. Wirth. A plea for lean software. *IEEE Computer*, 28(2):64–68, 1995.
44. M. Wooldridge and N. R. Jennings. Intelligent agents: Theory and practice. *The Knowledge Engineering Review*, 10(2):115–152, 1995.
45. M. Wooldridge, N. R. Jennings, and D. Kinny. The gaia methodology for agent-oriented analysis and design. *Journal of Autonomous Agents and Multi-Agent Systems*, 2000. to appear.
46. Frank Zappa. Joe’s garage. Munchkin Music, 1979.