From Procedures, Objects, Actors, Components, Services, to Agents

Jean-Pierre Briot

▶ **To cite this version:**

Jean-Pierre Briot. From Procedures, Objects, Actors, Components, Services, to Agents. Bertrand Meyer. The French School of Programming, Springer, pp.125-146, 2024, 10.1007/978-3-031-34518-0 . hal-03787494

HAL Id: hal-03787494
https://hal.science/hal-03787494v1

Submitted on 25 Sep 2022

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Chapter 1
From Procedures, Objects, Actors, Components, Services, to Agents
A Comparative Analysis of the History and Evolution of Programming Abstractions

Jean-Pierre Briot

Abstract The objective of this chapter is to propose some retrospective analysis of the evolution of programming abstractions, from *procedures*, *objects*, *actors*, *components*, *services*, up to *agents*, by replacing them within a general historical perspective. Some common referential with three axes/dimensions is chosen: *action selection* at the level of one entity, *coupling flexibility* between entities, and *abstraction level*. We indeed may observe some continuous quest for higher flexibility (through notions such as *late binding*, or *reification* of *connections*) and higher level of *abstraction*. Concepts of components, services and agents have some common objectives (notably, *software modularity and reconfigurability*), with multi-agent systems raising further concepts of *autonomy* and *coordination*, notably through the notion of *auto-organization* and the use of *knowledge*. We hope that this analysis helps to highlight some of the basic forces motivating the progress of programming abstractions and therefore that it may provide some seeds for the reflection about future programming abstractions.

1.1 Introduction

Object-oriented programming, software components and multi-agent systems are some examples of approaches for software design and development with significant impact. Both offer abstractions for organizing software as a combination of software elements, with a common objective of facilitating its *evolution* (first of all, replacement and addition of elements). In this chapter, our initial objective is to conduct a comparative analysis between software components and multi-agent systems (in the following, we will use terms, respectively, *components* and *agents*). In order to better compare them, we

Sorbonne Université, CNRS, LIP6, F-75005 Paris, France
e-mail: Jean-Pierre.Briot@lip6.fr

replace them within some general *historical perspective* of the *programming evolution* (taking some inspiration from [31]).

1.2 Related Work

There are various comparative studies between agents (and multi-agent systems) and, e.g., objects [47], concurrent objects [30, 31] and actors [39]. This article integrates some of these analyzes and complements them with the concepts of components and of services, which, to our knowledge, has not yet been the subject of such systematic comparative studies. Let us mention the organization in France in 2004 and 2006 of two successive workshops about multi-agent and components: "Journées multi-agents and composants" (JMAC), followed by a journal special issue [5].

Let us also cite here, for additional information, some comparative analyzes about different component models [21, 41] and about various multi-agent platforms and languages (based on object-oriented, logic or component-based models) [6].

1.3 Analysis

We have chosen a common conceptual *frame of reference* with *three dimensions* that we consider important issues in programming and software:

- *selection of the action* to be performed by an entity – This is about *when* and *how* an *entity* (a software entity, such as a procedure, a function, an object, an agent, etc., or a physical entity, such as a robot or an interconnected device) will *select* (decide) what action to be performed, through the activation of a corresponding code. The evolution of programming shows the need for deferring always *later* and *further* this decision (this has been coined as "*ever late binding*"). In addition, for an agent, such a decision may be based, not only on the nature of the invocation, as for classical programming languages, but also on the agent's own *knowledge* and context (e.g., by its *goals*), in a *proactive* and not only *reactive* manner;
- *flexibility of the coupling* between entities – This represents the ability to put in *relation* several software entities. The evolution of programming shows the need to represent and manipulate such relations independently of the implementation of the entities, in order to favor *adaptability* through some *explicit* manipulation of the relations. The concept of *software architecture* [56], i.e. the *assemblage* of components via explicit *connectors*, represents therefore a major advance. The concept of *service* brings further *dynamism* (via the concept of *discovery* of services) and *autonomy* for the entity itself (the selection of the actual service(s)). Multi-agent systems

raises the description of the coupling even further through concepts such as *organization* and *interaction protocol*.

- *level of abstraction* – This represents the expression level offered to the designer and to the programmer. We can observe a progressive quest for higher-level abstractions, from the initial low-level concepts of *instruction*, to abstract concepts of *procedure* and *abstract data types*, which turn out independent of an implementation platform, and finally up to *knowledge* concepts, such as *plan* and *intention*, upon which automated reasoning mechanisms can be applied.

It should be noted that these three dimensions are not completely independent: action selection may have some impact on coupling flexibility, and the choice of abstractions and mechanisms for action selection and for coupling are clearly related with the level of abstraction. In addition, it is possible (as, e.g., in [34]) to consider action selection and coupling uniformly, both based on a single mechanism: *binding*[1], which encompasses both: a) binding of the call to the effective code, in the case of action selection, and b) binding of a reference to another entity, in the case of coupling. However, we prefer to distinguish them, because their corresponding levels are conceptually distinct (*micro* versus *macro*), as well as their corresponding professions (programmer versus system architect).

Figure 1.1 illustrates our proposed 3-axes frame of reference. Each axis will be analyzed, respectively, in Section 1.4 (action selection), Section 1.5 (coupling flexibility) and Section 1.6 (abstraction level).

1.4 Action Selection

The first programming languages, e.g., the first version of Fortran, consider program behavior (code) and program state (data) within a common *global data space*. The different instructions are identified through their *line number*. The selection of the action (to be performed) is therefore expressed *globally* and *statically*.

Structured or *modular* programming languages, such as Pascal and then Modula, introduce some *modularization* of the code, expressed under the form of *procedures*. The selection of the action therefore gains in abstraction, the indication of the code to be executed being expressed via a *symbolic name* and no longer by a line number. However, the association of a name of a procedure to its corresponding code remains *static*. In some dual movement, data gradually gains structure and generality, thanks to the concept of *abstract data structures*.

[1] Note that *dynamic linking* [38] was probably introduced as early as in 1959 by the Multics multiuser operating system, in order to allow resolving external references at runtime.

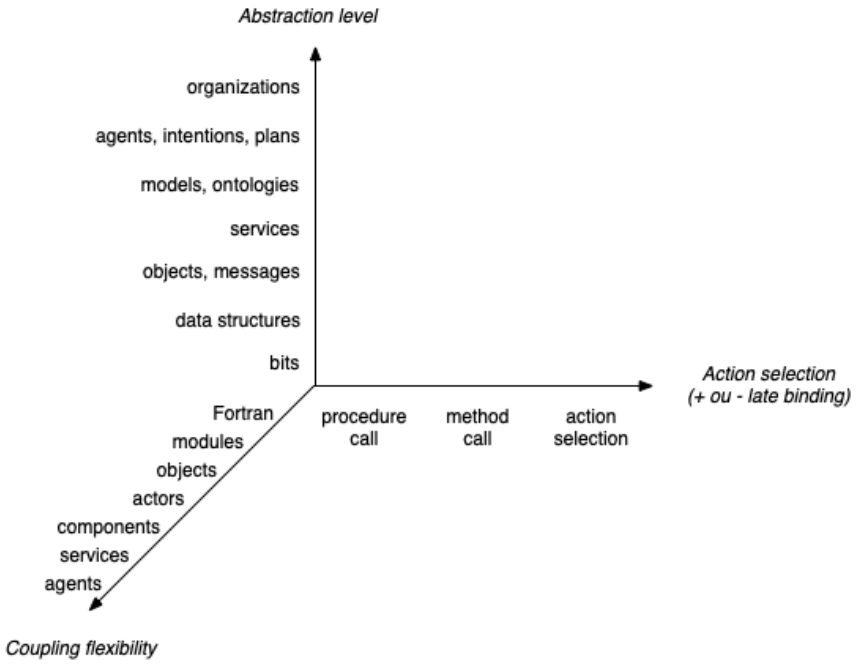

Fig. 1.1 Programming evolution

Object-oriented programming languages, with pioneers such as Simula 67 and then Smalltalk, bring some major innovation, through the reunion of some procedures and their associated data into a *self-contained* capsule, named an *object*. Data thus become *internal* and *private* to the object and its procedures (called *methods*) and *message sending* is the only way to invoke an object, which will activate one of its procedures.

Some decisive advance is the discipline of *late binding* such as in Smalltalk, i.e. the procedure to be invoked will be determined according to the *class*[2] of the actual object invoked, and not according to the declaration of the *type* of the variable that references it[3]. This means that the binding of the procedure, and therefore the selection of the action, is delayed at *runtime* and not statically resolved at compile time, such as in C++ early binding discipline. (Actually, C++ introduced virtual functions to partially alleviate this limitation, but this partial solution cannot benefit from further redefinitions

[2] A *class* is the definition of a family of *similar* objects. It is the class which defines the *methods* (procedures) and the *variables* (data model) common to the objects which will be its *instances*, i.e. created by/from it.

[3] We deliberately do not discuss here the relations between *binding* and *typing* (and subtyping), due to the fact that they are subtle and non consensual. For one analysis (among others), see, e.g., [15] and also the companion chapter by Giuseppe Castagna on types.

of methods once the library has been compiled, therefore in opposition to Bertrand Meyer's open-closed principle [42]).

The concept of *agent* introduces *internal autonomy* to the selection of the action. It is no more governed only externally by the nature of the request, as for a procedure or method call, but also *internally* by the internal state of the agent, since this may include be *cognitive* information of the agent such as its own *goals*. Therefore, an agent is no longer only *reactive* (to invocations) like objects, but also *proactive* [47]. Thus, the concept of action selection takes its full meaning, as for a robot or a human being, who can arbitrate his own action(s) at any given time, depending on both his own objectives and on information collected (messages from other agents or/and *perceptions* of the *environment*). *Arbitration* can be done at a symbolic level in *cognitive agents*, e.g., according to the agent *intentions*, in an architecture such as BDI [33]. *Reactive agents* have much simpler, *stimulus*-based action response mechanism, close to message response mechanism in object-oriented programming. Note that there is in fact some continuum between cognitive and reactive agents categories, with hybrid architectures attempting at reconciling and combining the two approaches (see, e.g., the InteRRaP hybrid architecture [44]). Last, some *sub-symbolic* mechanisms (without an explicit representation of the world) for regulation, often inspired by biology (metabolism, emotions, motivation, adaptation, see, e.g., [66]) can also be incorporated to agents.

Les Gasser proposed in 1998 as one of the fundamental concepts of agent programming the concept of *structured persistent action*, in which an agent is *autonomously* and *persistently* trying to accomplish something, independently of the way it is programmed [31]. In standard procedural programming, the programmer explicitly controls the attempts, while the concept of structured persistent action abstracts and encapsulates such a mechanism. More precisely, the designer provides the description of the objective or criteria for success, as well as in general a collection of methods and recipes, which the agent will select and control autonomously. Note that some similar mechanisms have already been proposed, for instance *declarative* programming and *backtrack* in logic programming languages such as Prolog, or the general concept of *search*. But, in our opinion, the concept of structured persistent action represents in an interesting way the encapsulation of: a notion of *choice*, *information*[4], an *iterative control structure* (of type repeat until), and proper *resources* (own process or thread). In addition, we consider the interaction of the agent with its environment to ensure some *feedback* over its actions and choices (e.g., through some reinforcement learning mechanism). Last, we may observe that the selection (and therefore the choice) of the action takes place at the moment of the action by the agent and not at the moment of the programming of the agent. Therefore, the concept of agent is situated within the quest for "*ever late binding*".

[4] Some information about the possible choices of actions related to the domain in which the agent acts. Such information can be symbolic (beliefs, models, plans...) or not, depending on the choice of agent architecture and of the representation of the world.

Table 1.1, inspired by [47], summarizes our analysis. The horizontal axis of Figure 1.1 illustrates the evolution of action selection within our frame of reference.

Programming	*Monolithic ex: Fortran*	*Modular ex: Pascal*	*Object-oriented ex: Java*	*Agent-Oriented ex: AgentSpeak*
Behavior	Global	Modular	Modular	Modular
State	Global	Modular and external	Modular and internal	Modular and internal
Invocation (and Selection)	Global and static *(goto)*	External and static *(procedure call)*	External and dynamic *(method call)*	Internal and external and dynamic *(ex: goal-driven)*

Table 1.1 Structure of entities and action selection

1.5 Coupling Flexibility

The modeling of the *coupling* between software entities is a fundamental aspect for the structuring of the software. It actually covers several facets:

- *structure*: the architectural concepts (e.g., *references*, *connectors*...) for the structural coupling between software entities;
- *communication*: the modes of communication between software entities, characterized mainly by: the mode for the *designation* of the receiver, the mode for *data transfer*, and the mode for *temporal coupling*.

1.5.1 Structural Coupling

The question of the *structural coupling* between software entities has been initially addressed by the notion of *reference* to an entity, through some means for identifying it (identifier). Therefore, one may designate a software entity (e.g., some data, object or function), in order to use it and to communicate its reference to other entities. This model, simple but effective and general, survived with object-oriented programming languages.

For instance, an object `A` references an object `B`, and thus will be able to send requests to `B`. In practice, the internal representation (implementation) of `A` includes a variable whose value is the identifier of object `B`. Changing a reference is easy, by just changing the value of the variable, for instance to the identifier of a third object `C`. However, we can observe that this modification can be done only *internally* to object `A`, the only one authorized to access its private data (following the encapsulation principle).

A serious limitation occurs when we want to *extend* a reference, for instance so that `A` refers *both* to `B` and to `C` (see the left part of Figure 1.2). Since a variable has only one value, this cannot be expressed directly. It is therefore necessary to introduce some data structure (a collection, e.g., a list), containing `B` and `C`. The message sending instruction must also be modified, by introducing an iterator on the collection. Overall, this implies the modification of the internal representation of object `A` (in other words, to reimplement it), whereas it is only a question of extending the reference and the coupling, initially from `A` to `B`, into from `A` to `B` *and* `C`.

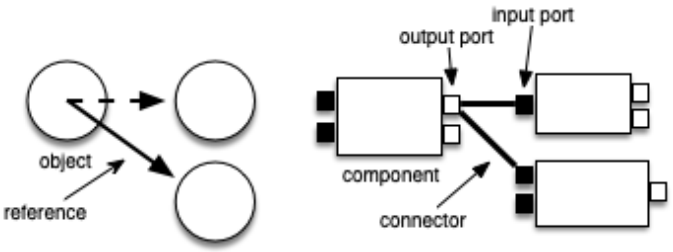

Fig. 1.2 Objects coupling versus components coupling

The concept of *software component* brings some notable improvement to this problem by *externalizing* the references, describing them as explicit *output interfaces*. Therefore, a component regains some *symmetry* at the level of interfaces between *input interfaces* (which are traditional for procedures and objects) and *output interfaces*, alternatively named, respectively, *provided interfaces* and *required interfaces*. (For a more complete analysis of the characteristics of software components and a comparison between different component models, see, e.g., [21] and [41].)

Coupling thus becomes *explicit*, *reified* (i.e. coupling is made into first class entities, the *connectors*) and *external* (to the software entities). Previous example is therefore achieved by the simple addition of a connector, as illustrated in the right part of Figure 1.2.

Note that a component can have multiple interfaces (input or/and output interfaces). To be able to identify them individually, an identifier, usually named a *port*, is associated to each interface. This is an important difference with an object which has only one identifier and entry point. An interesting consequence is that components are *compositional*. That is to say that a composition of several components is equivalent[5] to a component with the

[5] Actually, we should distinguish between *functional composition*, which is a simple assembly of components, and *structural composition*, which encapsulates a functional composition and identifies it as a new component, often referred to as a *composite component*. [21] analyzes their respective binding techniques, named *horizontal binding* and *vertical binding*. We believe that the concept of structural composition is important, as it provides *encapsulation* and *hierarchy*, which both proved to be useful to control complexity. How-

corresponding union of input ports and output ports. On the opposite, objects are not directly compositional: a composition of several objects is not immediately equivalent to an object, as it has more than one entry point.

Therefore, components provide an explicit architectural vision. The notion of *software architecture* [56] focuses on the logic of the coupling between the components, independently of their internal implementation. *Architecture description languages* (ADL) [56] are dedicated to the specification of the architecture of an application and they are indeed very different from standard programming languages. Information about the typing of component interfaces is used to verify correctness of the assembly, i.e. the conformity between the interfaces which are brought in relation. Different types of connectors are usually considered and correspond to different *architectural styles* (e.g, layered, pipes and filters, broadcast of events, etc. [56]) and their associated communication protocols. Connectors can also represent *non-functional properties* (such as distribution, quality of service, etc.) and therefore have their own semantics [2].

In order to express not only specifications about the types of data (typing information) but also about the *behavior* of components, notions of *contracts* have been proposed. For instance, [3] considers four successive levels of contracts: syntactic, behavioral, synchronization and quality of service. Depending on the case, they can be *guaranteed*, *verified* or *negotiated*. The syntactic level is based on a type system. The behavioral level is usually based on *assertions* (the three main types being: pre-conditions, postconditions, and invariants). But, compared to the use of assertions within a program, the idea of contracts is to specify them in a modular way and *visible* through the interfaces of a component, in order to be able to specify properties that can engage more than one software entity [43].

In addition, components introduced the idea of "ready to wear, to deploy, and to use", i.e. a component is some self-contained *unit of deployment*, with all its code and also its documentation [62].

The concept of *service* of *service-oriented architectures* (SOA) – including in particular *web services* [16] – brings dynamism to coupling, and moreover autonomy, via *discovery* and *dynamic selection* of other services (as shown in the left part of Figure 1.3). Coupling between entities is therefore no longer only managed by the designer of the application, but by the entities themselves (this corresponds to some degree of *self-organization*). For instance, an electronic travel agency service, looking for services to perform subtasks (e.g., flight reservation, hotels, etc.), will thus be able to *identify*, *select* (in general, according to various criteria, e.g., availability, price, flexibility, etc.), and *contract* sub-services. Therefore, services are subject to more or less elaborate descriptions, which are made available (*published*), e.g., through some directory of services, similar to telephone numbers yellow pages. For web ser-

---

ever, only a minority of component models support composite components (e.g., Fractal [13] and MALEVA [12], but not JavaBeans [61] nor CORBA Component Model (CCM) [50]).

vices, UDDI (Universal Description, Discovery and Integration) and WSDL (Web Services Description Language) standards [16] specify, respectively, directories and descriptions of services.

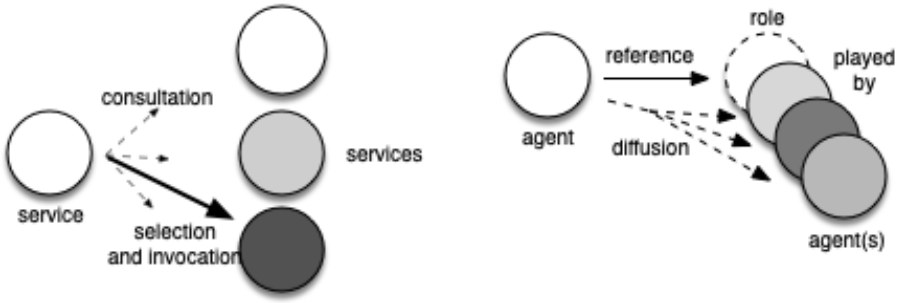

Fig. 1.3 Services coupling and agents coupling

Multi-agent systems further extend dynamism and autonomy by trading some *syntactic* coupling (following some typing discipline) for some *semantic* coupling, based on knowledge (via abstractions such as: *task*, *plan* and *intention*) and some *social organization of work* (via abstractions such as: *organization*, *role*, *norm* and *negotiation*).

An *organization* specifies the different *roles* constituting it (e.g., roles of producer, consumer and broker) and their *relationships* (e.g., dependency and hierarchy). A *role* can be *played* by one or more agents and the same agent can also possibly play more than one role simultaneously. Note that an agent referencing a role subsumes a reference to all the agents *fulfilling* (at the time of the interaction) this role, as illustrated in the right part of Figure 1.3. (This mechanism of *abstract role designation* of the receiver will be further analyzed in Section 1.5.2.1.)

Two important capacities of an organization are its *dynamism* and its *autonomy*. Some dynamic reorganization can be triggered: in a *top-down* manner, e.g., the reorganization of a robotic football team (as in the RoboCup contest [37]), according to a more defensive strategy on the initiative of the coach [36]; or in a *bottom-up* manner, e.g., with the dynamic formation (and then dissolution) of a micro-organization of type "one-two" on the initiative of some player agent [23]. Examples of abstract models of organizations are AGR [26] and MOISE+ [36].

As for services, multi-agent systems also often use various mechanisms for putting agents into relation: by some intermediary agents, directory agents, or facilitator agents guided by the content of the message (e.g., in KQML [27]); or by some selecting and contracting mechanism, as, e.g., the *contract net protocol* [59] (which will be introduced in Section 1.5.2.3).

To conclude, note that the software architectures and components communities started to support automatic reconfiguration, e.g., for nomadic appli-

cations [24]. But the knowledge and social-oriented approach of multi-agent systems is more ambitious, and therefore also more difficult to *verify*. We thus find out some classic dilemma between the growing needs for *flexibility*, through some delegation of initiative, and the needs to ensure some *guarantees* on the operability of the system.

1.5.2 Communication Coupling

The expression of the *mode of communication* between software entities includes several important characteristics (sub-facets). We consider here the three main ones:

- how to *designate the receiver(s)*, e.g., point to point, multi-point, indexed by content, via the environment, etc.;
- the mode for *data transfer*, e.g., unidirectional, bidirectional with value return, via a shared space, etc.;
- the *temporal coupling* (in other words, the way communications are *synchronized*), e.g., synchronous, asynchronous, with an anticipated (future) response, coordinated by a protocol, etc.

1.5.2.1 Designation of the Receiver

The mode of communication between objects is fundamentally *point to point*, i.e. one to one and with explicit designation of the receiver of the message. Components introduce *multi-point* communication, as an output of a component can be connected to more than one component. An interesting type of connector is the event broadcasting connector, corresponding to the *publish-subscribe* architectural style [56]. It offers an *indirect* and *dynamic* management of connections by the components themselves, through a mechanism of subscription of a component to the event broadcaster. (The subscription criteria and the distribution method may vary, see, e.g., the classification proposed in [25].) This type of mechanism became widespread (e.g., in applications based on standard objects) although it remains very representative of the concept of connector between software components, defined and manipulated externally to them (as it has been analyzed in Section 1.5.1).

The *shared spaces* (repositories) architectural style, illustrated by, e.g., blackboards and tuple-spaces (for instance, the LINDA model [32]), introduces a mode of designation of the receiver completely *implicit*, since it will be *indexed* by the actual *content* of the message. In this model, active entities (e.g., processes or agents) can insert and index structured data within the shared space. Data will be consumed *opportunistically* by active entities looking for the corresponding data patterns.

Services, and moreover multi-agent systems, generalize mechanisms of indirect and dynamic designation, through some contracting protocols or the consultation of broker or directories agents (as it has been presented in Section 1.5.1). Services or agents can therefore dynamically select their own *interlocutor*. Some more implicit mechanism is the notion of *facilitator*, guided by the content of the message [39] (e.g., in KQML [27], to be analyzed in Section 1.6.3). Another type is the abstract designation of a receiver through a *role*, as, e.g., in the AGR (agent group role) *organizational model* [26]. In such role-based models, agents usually designate some role (e.g., midfielder or striker, in a RoboCup football organization), rather than some specific agent, as the receiver of a communication. As a consequence, all the agents fulfilling this role at the time of communication will receive the information (see the right part of Figure 1.3).

Last, in certain types of multi-agent systems, in which the environment (physical or not) is explicitly modeled, the agents can communicate via the environment, though inserting specific data, for example *pheromones* for ant-based algorithms. (Such algorithms can be used as a general meta-heuristic optimization method, the environment having then no longer relation with a physical reality.) Note that, there is also some trend in multi-agent systems for promoting the environment as a first-class abstraction (for more details, see, e.g., [64]). There is also a similar trend for promoting entities without internal goals and characterized by a function as first-class entities named *artefacts*, which are manipulated (use, selection or construction) by agents [52].

1.5.2.2 Data Transfer

The mode for *data transfer* in object-oriented programming is *bidirectional*, with some *return of value* (unless the programmer explicitly specifies that there is no return value, e.g., in Java using the special data type `void` which represents the absence of data). It is inherited from the procedural or functional call. It corresponds (as we will see in Section 1.5.2.3) to a *synchronous* call, i.e. with the sender suspending its activity while waiting for the completion of the processing of the request by the receiver.

The *actor* model [35, 1] introduces some *unidirectional* (and *asynchronous*, see Section 1.5.2.3) data transfer mode. This was motivated by the concurrent and moreover distributed nature of the model, in order to avoid unnecessary and unbounded waiting for an acknowledgement of data transfer completion. Therefore, data transfer is carried out only *one-way* from the sender to the receiver. If the receiver wants to return a value, it must be done explicitly, by sending another message. Some languages based on actors, as for instance ABCL (Actor-Based Concurrent Language) [65], provide the programmer with a choice between a one-way asynchronous message send and a two-way synchronous call. (Note that the Actalk object-oriented framework offers in a

single pedagogical framework various types of actor-based and object-oriented concurrent programming abstractions, regarding various models of: action selection, activity, communication [7] and internal synchronization [8].)

Component models, such as CORBA component model (CCM) [50][6] often also propose these two modes of data transfer: bidirectional though a procedure call (via input and output interfaces, named *facets* and *receptacles* in CCM), and unidirectional though event diffusion (via event *sources* and *sinks*), see Figure 1.4.

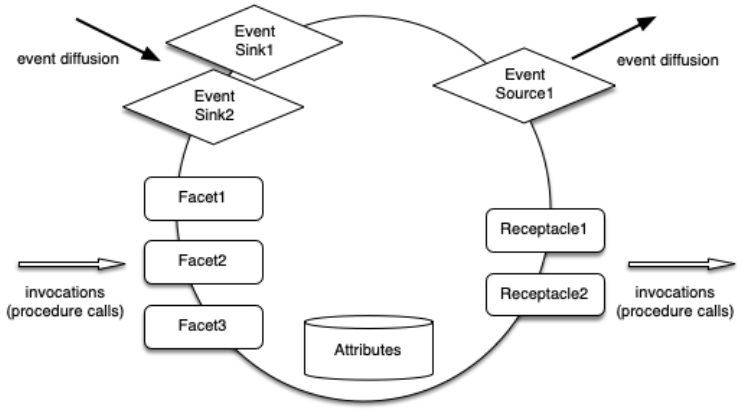

Fig. 1.4 CCM component model

The shared spaces architectural style (presented in Section 1.5.2.1) introduces a mode of data transfer, indirect, via some mediation structure and the distinction between production and consumption.

Services are generally based on simple invocation protocols, in particular the case of web services. One of the main reasons for the success of web services is likely their easy deployment on top of the widespread web infrastructure and its HTTP protocol. The SOAP protocol [16] (originally the acronym for "Simple Object Access Protocol") supports both bidirectional and unidirectional modes.

Multi-agent systems generally offer the unidirectional (and asynchronous) transfer mode of actors but expressed within more elaborate *agent communication languages*, which allow to specify with precision and details the nature of the information to be communicated (as it will be presented in Section 1.6.3).

Last, some possible communication via an environment (by adding, removing, or consuming data, as in the shared spaces architectural style, such, e.g.,

[6] Note that an example of a more recent component model (also an industry standard), also integrated into a service-oriented architecture, is CSA (Composite Services Architecture) [45]. However, we have chosen here to illustrate our analysis through the CCM model, for its historical and pedagogical value.

the LINDA model [32], presented in Section 1.5.2.1) represents some indirect mode of data transfer.

1.5.2.3 Temporal Coupling (Synchronization)

The original communication model between software entities (in a sequential and centralized world) is the procedural or functional call, with return of a value. The sender activity is *suspended* during the processing of the request by the receiver. A direct transposition into a concurrent setting sticks to these principles, with the sender waiting for the call to be completed – this is referred to as *synchronous* transmission. A direct transposition into a distributed setting is represented by the RPC (*Remote Procedure Call*), also synchronous.

The actor model introduces an *asynchronous* mode of communication as its foundation, i.e. without waiting for the message to be processed – and before that, to be received – by the receiver. Asynchronous communication is more appropriate to a concurrent or/and distributed setting (due to the potential latency of the communication network, this avoids waiting for the delivery of the message to the receiver, as well as its availability to process it). Therefore, the actor model assumes the existence of a *mailbox* for each actor, which will store the messages in the order of the arrival (FIFO type discipline). The actor model thus introduces some *temporal decoupling* between *sending*, *receiving*, *processing start*, and *processing completion* of the message. As explained in Section 1.5.2.2, some actor-based languages, such as ABCL, can provide both one-way asynchronous and two-way synchronous communication [65]. Another mode provided by actor languages and ABCL is a promise/anticipation of the response, often named *future*. It corresponds to *eager evaluation* (also coined as *wait by necessity* in [14]), the actual exact opposite of *lazy evaluation*. Scala is an example of a programming language which integrates functional, object-oriented, and actor programming [48]. Last, for an analysis about the different ways of mapping the object-oriented programming model to concurrent and distributed programming requirements, please refer, e.g., to [11].

Agent communication languages (ACL), in particular FIPA[7] ACL [28], allow the specification of a protocol associated with a communication. The *protocol* specifies the *coordination* of valid message exchanges between agents. Temporal coupling is therefore expressed in a relatively general manner and with an arbitrary number of messages and agents. Example of families of agent protocols are: interaction (e.g., inform, request, deny...), coordination (e.g., simple or iterated call for proposals, see next paragraph) and auction

[7] FIPA is the acronym for the Foundation for Intelligent Physical Agents, an IEEE Computer Society standards organization which promotes agent-based technology and the interoperability of its standards with other technologies [29].

(e.g., English or Dutch, with, respectively, increasing or decreasing initial price).

A classic example of a multi-agent protocol is the call for proposals (also named the *contract net protocol*). Figure 1.5 shows the corresponding interaction diagram (as specified by FIPA [28]). Successive phases are:

- the broadcast of the initial call (where `cfp` stands for call for proposals) by the *initiator* (also named *contractor*) to the participants;
- various proposals (or refusals) made by the participants, controlled by some deadline (timeout) for responding;
- the selection and acceptance (or rejection) of a proposal by the initiator; and finally
- the communication by the selected participant (also named *sub-contractor*) about the finalization and the result (or the failure) to process its proposal.

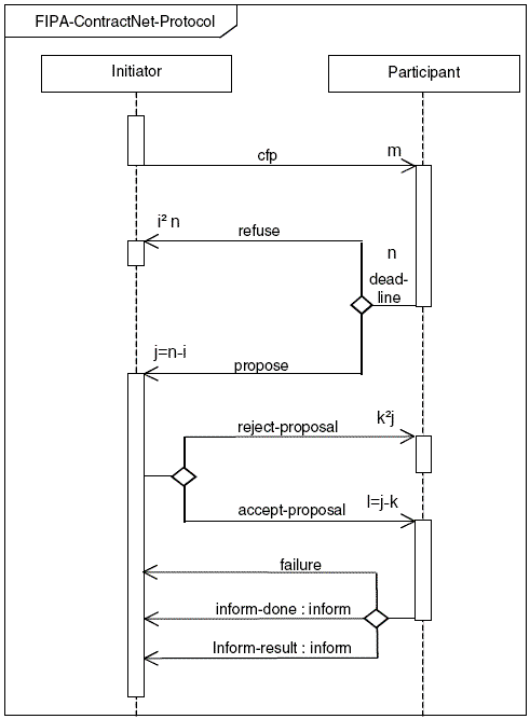

Fig. 1.5 Contract net protocol. Figure reproduced from FIPA Contract Net Interaction Protocol Specification, Foundation for Intelligent Physical Agents, 2002.

Web services also offer analog coordination mechanisms, also named *choreography*. The Web Services Choreography Description Language (WS-CDL) has been initially defined with this intent by the W3C (World Wide Web

Consortium standard [63]). It has since been replaced by the BPEL (Business Process Execution Language) and BPMN (Business Process Model and Notation) standards [46]. (We will not detail here the characteristics of services and Web services, which are the subject of standards and numerous technical specifications, because that would be the subject of another article. See, e.g. [16] and [53], as well as [54] for an agent perspective on web services.)

Table 1.2 summarizes the evolution of coupling according to the 2 main facets: structure and communication, the latter one with its 3 sub-facets: designation of the receiver(s), data transfer mode, and temporal coupling (synchronization).

The diagonal axis of Figure 1.1 illustrates the evolution of coupling flexibility within our frame of reference. Note that the evolution of coupling flexibility is not completely linear: actors have been proposed before components but their respective main focuses are different (respectively, concurrency and architecture); web services have been proposed after multi-agent systems.

Coupling	*Objects*	*Actors*	*Components*	*Services*	*Agents*
Structure	Implicit internal *(references)*	Implicit internal *(references)*	Explicit external *(connectors)*	Implicit volatile *(invocations)*	Implicit external *(roles)*
Communication					
Receiver(s) designation	Point to point explicit	Point to point explicit	Multi-point explicit or implicit *(publish-subscribe)*	Multi-point dynamic *(discovery and selection)*	Multipoint explicit or implicit *(role designation)*
Data Transfer	Bi-directional *(value return)*	Uni-directional	Bi- or uni-directional *(events)*	Bi- or uni-directional	Uni-directional direct or indirect *(via environment)*
Synchronization	Synchronous	Asynchronous	Synchronous or asynchronous	Synchronous or asynchronous	Asynchronous or protocol

Table 1.2 Nature of the coupling

1.6 Abstraction Level

The history of programming begins with concepts very close to the machine (instructions, integers, etc.), then progressively identifies some higher level *abstractions* (procedure, function, data structure, semaphore, process, object, message, component, model, etc.). The concepts of agent and organization

continue this evolution towards more abstraction as well as towards more explicit knowledge.

1.6.1 From Data to Concepts

The transition from primitive data types to abstract data types allows the modeling and naming of arbitrary classes of objects. Object-based programming introduces some major evolution step, with objects modeling and representing (reifying) conceptual or physical objects of the application domain considered [55]. In other words, we moved from data to *concepts*. Agents will extend this evolution with an explicitation of the domain (including human) *knowledge*. Cognitive agents introduce the notion of *mental state*, inherited from symbolic artificial intelligence (see, e.g., [57]), with some symbolic representation of cognitive concepts, such as: belief, goal, desire, intention, etc. Furthermore, such internal knowledge of an agent can be communicated to other (external) agents, e.g., communication of beliefs, plans or/and intentions, in order for agents to learn about each others or/and coordinate their actions. Agents can also reason about the *context* (as coined in [20], "context is key") for *context-aware* applications such as ambient intelligence. ([17] identifies four basic types of context: *computational* context (i.e. state of resources of the device and of the network), *user* context (i.e. persons, places, or/and objects), *physical* context (e.g., luminosity, noise, or/and temperature) and *temporal* context (e.g., hour, day, or/and period of the year).)

The object-oriented discipline of message sending also provides some self-documentation, as the subject and the request type are specified explicitly. Agent communication languages raise further the explicitness of information and knowledge. Indeed, information that had remained implicit (and hidden) in object-oriented and component-based applications – such as intention of communication, coordination logic, plans, etc. – and remained in the mind of the programmer, become explicit and thus better document the program. Moreover, this information could also be used by the agents themselves (for example to coordinate, reason about communication failures, replan, reorganize, etc.).

1.6.2 Reification

An additional approach, transversal to the type of abstractions proposed (objects, components..., agents), is *reification*. It is the process by which an abstract concept about a computer program is turned into an explicit entity created in the programming language. In other words, something that was previously implicit and unexpressed is explicitly formulated at the level of the

language (thus often coined as "making something a first-class citizen"), and therefore made available to *inspection* and *manipulation*. The Lisp programming language has been a true pioneer, with its uniform vision of considering programs as data. This has been developed further in the Smalltalk programming language, which reifies various types of program and implementation entities, such as messages, contexts, classes, as actual Smalltalk objects. Static concepts, e.g., classes, are reified as permanent objects. (A class is thus an instance of a class, usually named a *metaclass*.) Computational concepts, like stack contexts and messages, are only reified on demand (e.g., in case of errors), for obvious efficiency reasons. The inverse operation, making a reified information (back) into an actual implementation, is named *reversion*, or *reflection*. An example in Smalltalk occurs in case of an error: the interactive debugger opens up and allows inspection of a reified context of the currently stopped computation. Once correcting the error, the debugger reinstalls the corrected computational context and resumes computation.

In addition, in some languages, an entity already explicit at the programming level, e.g., an object, may gain some explicit representation of some of its implementation characteristics, usually coined as its (or one of its) *meta-object*. Such types of self-described and introspective languages or architectures are usually named either *reflective architectures* or *meta-level architectures*. Indeed, this way of opening up implementations (into manipulable abstractions) in order to make them adaptable at some high level of abstraction turned out being very useful. See, e.g., [19] for some survey of reflective, meta-level and/or meta-object architectures, [10] and [18] for an example of minimal reflective object-oriented architecture, and [40] for a very developed one.

1.6.3 Interoperability Languages

Let us now examine *interoperability middleware*, which specifies and standardizes the exchange of information. CORBA object-oriented middleware designed by OMG [49] standardizes, through an *interface description language* (IDL), the types of data exchanged. The analogue for agents further refines the way information is exchanged. The IDL of CORBA is substituted (see details in next paragraph) by a more general agent communication language (ACL). In addition to the specific content of the message, an ACL communication can specify:

- *performative*: some symbolic designation of the *intention* of the communication (e.g., inform, deny, recruit, etc.);
- *content description language*: the language used to describe the content. It can be some programming language (e.g., Java) or some knowledge representation language (e.g., KIF, or SL [28]);

- *ontology*: the ontology(s) (i.e. some representation of a set of concepts, their properties and their relations) of the concepts referred to by the message (e.g., some standard ontology about transport and tourist services, for some electronic travel agency application);
- *protocol*: the protocol used for the communication (e.g., a call for proposals, named FIPA-Contract-Net, see Figure 1.5).

It should be noted that CORBA and ACL do not actually play exactly the same roles [60]. CORBA, through its IDL, provides some standard for specifying the interfaces (signatures) of objects and components. It also provides mappings (named *projections*) of this IDL in different programming languages (e.g., Java, Smalltalk, C++, etc.). Therefore, CORBA can automatically generate *implementation skeletons* for the calling party code and for the called party code, and thus ensure the translation and transfer of data. An ACL does not offer some standard for specifying interfaces of agents, but offers *instead* some general standard for specifying various properties of communication between agents, which is different. As listed above, ACL standardizes various properties such as intention, ontology and protocols. The first historically is KQML [27], followed by FIPA ACL [28].

1.6.4 Organizational Design

It is also important to highlight the preponderant role of the *design* of multi-agent systems. It is guided by the *organization of work* (through concepts such as organization, role, dependence, and *norms*) and by *knowledge* (mind states, such as belief and intentions), rather than by the *operational means* for achieving this work, which corresponds to the traditional procedural approach of programming (through data and procedures). Multi-agent *methodologies* (e.g., such as the pioneering Cassiopée [23]) often start with some analysis of organizations, roles and their dependencies, while considering separately (and later) implementation questions (such as: which agents will fulfill the roles, depending on what decomposition of tasks). Some agent-oriented design can then be carried out (implemented) in some multi-agent architecture, or through objects, actors, or/and components, the agent level not always appearing completely at the implementation level. However, keeping abstractions, such as agents and organizations, as entities explicitly represented at the *execution level*, offers of course possibilities of dynamic manipulation by the programmer, but above all by the entities (agents and organizations) themselves, thus offering possibilities of self-adaptation and self-organization (see, e.g., the organizational model MOISE [36]).

Finally, in the evolution and the elevation of programming abstractions, we also need to mention about *model driven engineering*, such as the *model driven architecture* (MDA) proposed by the OMG [51]), as a modeling level for the partial automation of the construction of applications. (For more details, see,

e.g., the companion chapter by Jean-Marc Jézéquel on modeling.) Note that this line of research is somehow orthogonal to a specific programming model (object-oriented, component-based, agent-oriented, etc.). There are efforts to couple multi-agent programming and model engineering, see, e.g., [58].

The vertical axis of Figure 1.1 illustrates the evolution of abstraction level within our frame of reference.

1.7 Conclusion

Due to the increasing needs for auto-adaptation of future distributed applications (such as, e.g., Internet of Objects), models of software components and software architectures are gradually gaining in terms of abstraction as well as in (self) adaptation and reconfiguration capacities (see, e.g., [60], and also IBM proposal for autonomic systems [4]). They get inspiration from multi-agent systems abstractions, while often relying on light-weight infrastructures such as, or inspired by, web services. The technology of web services is indeed simpler and lighter to implement and to deploy than some distributed component models (such as, e.g., CORBA), as current web infrastructure is sufficient. Web services provide the specification of the coordination between services (named choreography) although it does not yet reach the level of sophistication of multi-agent systems (on this topic, see, e.g., a comparative analysis of web services and agents [54]). Additional abstractions from distributed programming (such as, e.g., replication, groups and consensus, to manage fault-tolerance) and/or ambient programming (e.g., ambient references [22], to manage volatile connexions), are also needed to deal with distribution, fault-tolerance, volatility and uncertainty. (Note that distribution was not the focus of this study, for more details, see, e.g., the companion chapter by Michel Raynal on distributed programming.)

An important stake is therefore to be able to integrate and reuse, as much as possible, respective abstractions and experience from various programming models and communities. However, cultural specificities sometimes lead to some ignorance about respective works. One of the objectives of this analysis was therefore to (humbly) contribute to highlight some of the basic forces motivating the progress of programming abstractions, in order to favor mutual awareness, as well as possible cross-fertilization[8], and to provide some inspirational seeds about future programming abstractions.

[8] In that respect, the last section of an article in french [9] which was the basis for this chapter identifies various potential mutual cross contributions between software components and multi-agent systems: from agents to components, e.g., by using mapping and negotiation techniques to assist the assemblage of components; and from components to agent(s), e.g., to structure and modularize its architecture.

Acknowledgements

The premises of this study go back to an interview that we conducted with Les Gasser on the relationship between objects and agents, published in a special series on actors and agents [31]. We thank him for his pioneering and fundamental contribution to this reflection and we dedicate this article to his memory. We also would like to thank Pierre Cointe and Jean-François Perrot for having formed us to the magic of programming.

References

1. Agha, G.: Actors: a Model of Concurrent Computation in Distributed Systems. Series in Artificial Intelligence. MIT Press (1986)
2. Allen, R., Garlan, D.: Formal connectors. Research Report CMU-CS-94-115, Department of Computer Science, Carnegie Mellon University, Pittsburgh, PA, USA (1994)
3. Beugnard, A., Jézéquel, J.M., Plouzeau, N., Watkins, D.: Making components contract aware. IEEE Computer **32**(7), 38–45 (1999)
4. Bigus, J., Schlosnagle, D., Pilgrim, J., Mills, W., Diao, Y.: Able: A toolkit for building multiagent autonomic systems. IBM Systems Journal **41**(3), 350–371 (2002)
5. Boissier, O.: (editor) Composants et systèmes multi-agents. L'Objet **12**(4) (2006)
6. Bordini, R., Dastani, M., Dix, J., Seghrouchni, A.E.F., Gomez-Sanz, J., Leite, J., O'Hare, G., Pokahr, A., Ricci, A.: A survey of programming languages and platforms for multi-agent systems. Informatica **30**(1), 33–44 (2006)
7. Briot, J.P.: Modélisation et classification de langages de programmation concurrente à objets : l'expérience Actalk. In: Actes du Colloque Langages et Modèles à Objets (LMO 94), pp. 153–165. INRIA/IMAG/PRC-IA, Grenoble, France (1994)
8. Briot, J.P.: An experiment in classification and specialization of synchronization schemes. In: K. Futatsugi, S. Matsuoka (eds.) Object Technologies for Advanced Software (ISOTAS 96), no. 1049 in LNCS, pp. 227–249. Springer, Kanazawa, Japan (1996)
9. Briot, J.P.: Composants et agents : évolution de la programmation et analyse comparative. Technique et Science Informatiques (TSI) **33**(1-2), 85–115 (2014)
10. Briot, J.P., Cointe, P.: A uniform model for object-oriented languages using the class abstraction. In: J. McDermott (ed.) 10th International Joint Conference on Artificial Intelligence (IJCAI'87), vol. 1, pp. 40–43. Morgan-Kaufmann, Milano, Italy (1987)
11. Briot, J.P., Guerraoui, R., Löhr, K.P.: Concurrency and distribution in object-oriented programming. Computing Surveys **30**(3), 291–329 (1998)
12. Briot, J.P., Meurisse, T., Peschanski, F.: Architectural design of component-based agents: A behavior-based approach. In: R.H. Bordini, M. Dastani, J. Dix, A.E.F. Seghrouchni (eds.) Programming Multi-Agent Systems - ProMAS 2006, no. 4411 in LNCS, pp. 73–92. Springer (2007)
13. Bruneton, E., Coupaye, T., Leclerc, M., Quéma, V., Stefani, J.B.: An open component model and its support in Java. In: 7th International Symposium on Component-Based Software Engineering, no. 3054 in LNCS, pp. 7–22. Springer (2004)
14. Caromel, D.: Toward a method of object-oriented concurrent programming. Communications of the ACM (CACM) **36**(9), 90–102 (1993)
15. Castagna, G.: Covariance and contravariance: Conflict without a cause. ACM Transactions on Programming Languages and Systems (TOPLAS) **17**, 431–447 (1995)
16. Chauvet, J.M.: Services Web avec SOAP, WSDL, UDDI, et XML. Eyrolles (2002)

17. Chen, G., Kotz, D.: A survey of context-aware mobile computing research. Technical Report TR2000-381, Department of Computer Science, Dartmouth College, Hanover, NH, USA (2000)
18. Cointe, P.: Metaclasses are first class: The ObjVlisp model. In: Conference Proceedings on Object-Oriented Programming Systems, Languages and Applications (OOPSLA '87), pp. 156–162. ACM (1987)
19. Cointe, P. (ed.): Meta-Level Architectures and Reflection – Second International Conference, Reflection'99 Saint-Malo, France, July 19-21, 1999 Proceedings. No. 1616 in LNCS. Springer (1999)
20. Coutaz, J., Crowley, J.L., Dobson, S., Garlan, D.: Context is key. Communications of the ACM **48**(3), 49–53 (2005)
21. Crnković, I., Sentilles, S., Vulgarakis, A., Chaudron, M.R.V.: A classification framework for software component models. IEEE Transactions on Software Engineering **37**(5), 593–615 (2011)
22. Cutsem, T.V., Dedecker, J., Mostinckx, S., Gonzalez, E., D'Hondt, T., Meuter, W.D.: Ambient references: Addressing objects in mobile networks. In: OOPSLA '06: Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications, pp. 986–997. ACM (2006)
23. Drogoul, A., Collinot, A.: Applying an agent-oriented methodology to the design of artificial organisations: a case study in robotic soccer. Journal of Autonomous Agents and Multi-Agent Systems (JAAMAS) **1**(1), 113–129 (1998)
24. Dubus, J., Merle, P.: Vers l'auto-adaptabilité des architectures logicielles dans les environnements ouverts distribués. In: M. Oussalah, F. Oquendo (eds.) 1ère Conférence francophone sur les Architectures Logicielles (CAL 2006). Hermès/Lavoisier, Nantes, France (2006)
25. Eugster, P., Felber, P., Guerraoui, R., Kermarrec, A.M.: The many faces of publish/subscribe. ACM Computing Surveys **35**(2), 114–131 (2003)
26. Ferber, J., Gutknecht, O.: A meta-model for the analysis and design of organizations in multi-agent systems. In: 3rd International Conference on Multi-Agent Systems (ICMAS 98), pp. 128–135. IEEE, Paris, France (1998)
27. Finin, T., Labrou, Y., Mayfield, J.: KQML as an agent communication language. In: J. Bradshaw (ed.) Software Agents, pp. 291–316. MIT-Press (1997)
28. FIPA: Agent Communication Language Specifications (accessed: 13/08/2021). http://www.fipa.org/repository/aclspecs.html
29. FIPA: Foundation for Intelligent Physical Agents (accessed: 13/08/2021). http://www.fipa.org/
30. Gasser, L., Briot, J.P.: Object-based concurrent programming and distributed artificial intelligence. In: N.M. Avouris, L. Gasser (eds.) Distributed Artificial Intelligence: Theory and Praxis, pp. 81–107. Kluwer (1992)
31. Gasser, L., Briot, J.P.: Agents and concurrent objects. IEEE Concurrency **6**(4), 74–81 (1998). Interview of Les Gasser by Jean-Pierre Briot
32. Gelernter, D., Carrierro, D.: Coordination languages and their significance. Communications of the ACM **35**(2) (1992)
33. Georgeff, M., Pell, B., Pollack, M., Tambe, M., Wooldridge, M.: The belief-desire-intention model of agency. In: 5th International Workshop on Intelligent Agents V: Agent Theories, Architectures, and Languages (ATAL'98), no. 1555 in LNCS, pp. 1–10. Springer (1999)
34. Ghezzi, C., Picco, G.: An outlook on software engineering for modern distributed systems. In: Monterey Workshop on Radical Approaches to Software Engineering. Venezia, Italy (2002)
35. Hewitt, C.: Viewing control structures as patterns of passing messages. Artificial Intelligence **8**(3), 323–364 (1977)
36. Hübner, J.F., Sichman, J.S., Boissier, O.: Developing organised multiagent systems using the MOISE+ model: programming issues at the system and agent levels. International Journal of Agent-Oriented Software Engineering (IJAOSE) **1**(3–4), 370–395 (2007)

37. RoboCup Federation Inc: RoboCup (accessed: 13/08/2021). https://www.robocup.org/
38. Janson, P.A.: Dynamic linking and environment initialization in a multi-domain process. ACM SIGOPS Operating Systems Review **9**(5), 43–50 (1975)
39. Kafura, D., Briot, J.P.: Introduction to actors and agents. IEEE Concurrency **6**(2), 24–29 (1998)
40. Kiczales, G., des Rivieres, J., Bobrow, D.G.: The Art of the Metaobject Protocol. MIT Press (1991)
41. Lau, K.K., Wang, Z.: Software component models. IEEE Transactions on Software Engineering **33**(10), 709–724 (2007)
42. Meyer, B.: Object-Oriented Software Construction. Prentice Hall (1988)
43. Meyer, B.: Applying design by contract. IEEE Computer **25**(10), 40–51 (1992)
44. Müller, J.P., Pischel, M.: The agent architecture InteRRaP: Concept and application. Technical Report RR-93-26, DFKI, Saarbrücken, Germany (1993)
45. OASIS: Open composite services architecture (CSA). Tech. rep., OASIS (Organization for the Advancement of Structured Information Standards), http://www.oasis-opencsa.org (accessed: 13/08/2021)
46. OASIS: Web services business process execution language (BPEL). Tech. rep., OASIS (Organization for the Advancement of Structured Information Standards), http://bpel.xml.org (accessed: 13/08/2021)
47. Odell, J.: Objects and agents compared. Journal of Object Technology (JOT) **1**(1) (2002)
48. Odersky, M., Spoon, L., Venners, B.: Programming in Scala. Artima (2010)
49. OMG: Common object request broker architecture (CORBA). Tech. rep., Object Management Group (OMG), http://www.omg.org/corba/ (accessed: 13/08/2021)
50. OMG: Corba component model (CCM). Tech. rep., Object Management Group (OMG), http://www.omg.org/technology/documents/formal/components.htm (accessed: 13/08/2021)
51. OMG: Model driven architecture (MDA). Tech. rep., Object Management Group (OMG), http://www.omg.org/mda/ (accessed: 13/08/2021)
52. Omicini, A., Ricci, A., Viroli, M.: Artifacts in the A&A meta-model for multi-agent systems. Journal of Autonomous Agents and Multi-Agent Systems (JAAMAS) **17**(3), 432–456 (2008)
53. Papazoglou, M.: Web Services & SOA, Principles and Technology, Second Edition. Pearson (2012)
54. Payne, T.: Web services from an agent perspective. IEEE Intelligent Systems **23**(2), 12–14 (2008)
55. Perrot, J.F.: Objets, classes et héritage : Définitions. In: R. Ducournau, J. Euzenat, G. Masini, A. Napoli (eds.) Langages et modèles à objets – État des recherches et perspectives, Collection Didactique, pp. 3–31. INRIA (1998)
56. Shaw, M., Garlan, D.: Software Architectures – Perspective on an Emerging Discipline. Prentice Hall (1996)
57. Shoham, Y.: Agent oriented programming. Artificial Intelligence **60**(1), 51–92 (1993)
58. Silva, V., Choren, R., Lucena, C.: Using UML 2.0 activity diagram to model agent plans and actions. In: International Conference on Autonomous Agents and Multi-Agent Systems (AAMAS 2005). Utrecht, The Netherlands (2005)
59. Smith, R.: The contract net protocol: High-level communication and control in a distributed problem solver. IEEE Transactions on Computers **29**(12), 1104–1113 (1980)
60. van Splunter, S., Wijngaards, N., Brazier, F., Richards, D.: Automated component-based configuration: Promises and fallacies. In: AISB 2004 Convention 4th Symposium on Adaptive Agents and Multi-Agent Systems (AAMAS-4), pp. 130–145. Leeds, UK (2004)
61. Sun: Javabeans specification. Tech. rep., Sun Microsystems Inc., http://java.sun.com/products/javabeans/ (2006)

62. Szyperski, C., Gruntz, D., Murer, S.: Component software: beyond object-oriented programming. Pearson Education (2002)
63. W3C: World Wide Web Consortium (accessed: 13/08/2021). https://www.w3.org/
64. Weyns, D., Parunak, H.V.D., Michel, F., Holvoet, T., Ferber, J.: Environments for multiagent systems – state-of-the-art and research challenges. In: D. Weyns, H.V.D. Parunak, F. Michel (eds.) Environments for Multi-Agent Systems – First International Workshop, E4MAS 2004, New York, NY, July 19, 2004, Revised Selected Papers, no. 3374 in LNAI, pp. 1–47. Springer (2005)
65. Yonezawa, A., Briot, J.P., Shibayama, E.: Object-oriented concurrent programming in ABCL/1. Sigplan Notices **21**(11), 258–268 (1986). Special Issue. Proceedings of the Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 86), Portland OR, USA
66. Ziemke, T., Balkenius, C., Hallam, J. (eds.): From Animals to Animats 12 – 12th International Conference on Simulation of Adaptive Behavior, SAB 2012, Odense, Denmark, August 2012, Proceedings. No. 7426 in LNAI. Springer (2012)