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

Jean-Pierre Briot[†]

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

Abstract: The objective of this chapter¹ 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 at highlighting 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 Introduction

Object-oriented programming, software components [4] and multi-agent systems [12, 24] 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*². In order to better compare them, we replace them within some general *historical perspective* of the *programming evolution* (taking some inspiration from [30]).

There are various comparative studies between agents (and multi-agent systems) and, e.g., objects [42], concurrent objects [29, 30] and actors [36]. This article integrates some of these analyzes and complements them the concepts of components and of services, which, to our knowledge, has not yet been the subject of such systematic comparative studies³. Let us also cite here, for additional information, two comparative analyzes about different component models [20, 37] and about various multi-agent platforms and languages (based on object-oriented, logic or component-based models) [7, 8].

2 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 indicates when and how a software⁴ entity 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;

 $^{^{1}}$ This article has been submitted to a project of book about the French school of programming, coordinated by Bertrand Meyer.

²In the following, we will use terms, respectively, *components* and *agents*.

³An initiative on the relations between components and multi-systems agents was the organization in France of two successive editions of Workshop "Journées multi-agents and components" (JMAC) in 2004 and 2006, followed by a journal special issue [6]. Note that this chapter is an adaptation and revision of an original article in french [11].

⁴or physical, in the case of a robot.

- 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 dynamicity as well as the explicit manipulation of the relations. The concept of software architecture [54], assemblage of components via explicit connectors, represents therefore a major advance. The concept of service brings further dynamicity (via the concept of discovery of services) and autonomy for the entity itself (the selection of the actual service(s)). Multi-agent systems bring a step further and higher the reification of the architecture and of the discipline of interaction through the concepts of organization and of 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, 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 abstraction. In addition, it is possible to consider action selection and coupling uniformly, both based on a single mechanism: $binding^5$: a) binding of the call to the effective code, in the case of action selection, and b) binding of a link to another entity, in the case of coupling. However, we prefer to *distinguish* them, because their corresponding levels are conceptually distinct (*micro* versus *macro* vision), as well as their corresponding professions (*programmer* versus *system architect*), and their corresponding abstractions and mechanisms (e.g., *agent architecture* versus *interaction protocol*).

3 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, bring 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*.

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 $class^6$ of the actual object invoked, and not according to the declaration of the type of the variable that references it⁷. This means that the binding of the procedure, and therefore the selection of the action, is delayed at vartime and not statically resolved at vartime as in C++ vartime vartime and not statically resolved at vartime vartime

Software *components* introduce the concept of "ready to wear, to deploy, and to use". As opposed to an object, which is orphan and potentially inoperative without its class as well as its parent class (superclass) hierarchy⁸, a component is *self-contained*, with all its code and also its documentation [48]. Therefore, on the contrary of an object, a component does not require any additional external information in order to select and process an action.

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, or more exactly by its $knowledge^9$, 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 [42]. Thus, the concept of action selection takes its full meaning, as for a robot or a human being, who can

⁵See, e.g., [33].

⁶A class is the definition of a family of *similar* objects. It is the class that defines the *methods* (procedures) and the *variables* (data model) common to the objects which will be its *instances*, i.e. created by/from it.

⁷We deliberately do not discuss here the relations between *binding* and *typing* (and *sub-typing*), due to the fact that they are subtle and non consensual. For one analysis (among others), see, e.g., [17].

⁸For example, in the case of an object migrating to another site which would not have already loaded its associated class hierarchy.

⁹The *knowledge* of an agent may be defined as what an agent *knows* about its *world* (including itself and other agents), this information being described through *interpretable concepts* (i.e., with the potential to be able to *reason* about them).

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

Table 1: Structure of entities and action selection

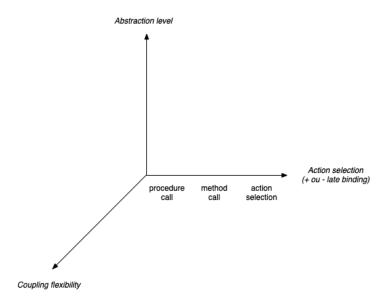


Figure 1: Evolution of action selection

arbitrate his own action(s) at any given time, depending on both its 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 [32]. 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 [39]). 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., [64]) can also be incorporated to agents.

Reflecting on the evolution of action selection, Les Gasser proposed in 1998 as one of the fundamental concepts of agent programming the concept of structured persistent action, in which the agent is autonomously and persistently trying to accomplish something, independently of the way it is programmed [30]. 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, that 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, informations of 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".

Table 1, inspired by [42], summarizes our analysis and Figure 1 illustrates it within our proposed frame of reference.

¹⁰Some informations 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 representation of the world.

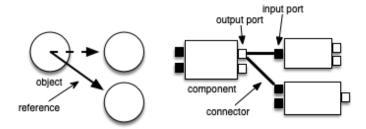


Figure 2: Objects coupling versus components coupling

4 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.

4.1 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 entity¹¹, 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 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.

The concept of $software\ component^{12}$ 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 $input\ interfaces^{13}$ and $output\ interfaces^{14}$.

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 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 equivalent¹⁵ to a component with the corresponding union of input ports and output ports. Otherwise, objects are *not* directly

 $^{^{11}}Simple\ data$ in early programming languages, functions in functional programming languages, objects in object-oriented programming languages...

¹²For a more complete analysis of the characteristics of software components and a comparison between different component models, please see, e.g., [20] or/and [37].

¹³which are traditional for procedures and objects.

 $^{^{14}}$ Alternatively named, respectively, provided interfaces and required interfaces.

¹⁵Actually, we must 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. [20] analyzes their respective binding techniques, named horizontal binding and vertical binding. We believe that the concept of structural composition is important [53], as it provides encapsulation and hierarchy, which both proved to be useful to control complexity. However, only a minority of component models support composite components (e.g., Fractal [15] and MALEVA [14], but not JavaBeans [59] nor CORBA Component Model (CCM) [45]).

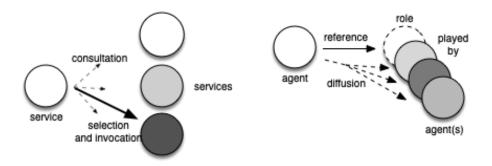


Figure 3: Services coupling and agents coupling

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 vision¹⁶. Architecture description languages (ADL) [54] are dedicated to the specification of the architecture of an application and they are indeed very different from standard programming languages. Informations about the typing of component interfaces are 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...[54]) 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 [3].

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, [5] 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 [38].

The concept of service of service-oriented architectures architectures (SOA)¹⁷ extends coupling with dynamicity, and moreover autonomy, via discovery and dynamic selection of other services (as shown in the left part of Figure 3). Coupling between entities is therefore no longer only managed by the designer of the application, but by the entities themselves, i.e., through 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, select¹⁸, and contract sub-services. Therefore, services are subject to more or less elaborate descriptions, which are made available (published), e.g., through directory of services, similar to telephone numbers yellow pages. For web services, UDDI (Universal Description, Discovery and Integration) and WSDL (Web Services Description Language) standards [18] specify, respectively, directories and descriptions of services.

Multi-agent systems further extend dynamicity 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 role¹⁹ (see the right part of Figure 3).

Two important capacities of an organization are its *dynamicity* and its *autonomy* (self-organization and self-reorganization). Some dynamic reorganization can be triggered: in a top-down manner, e.g., the reorganization of a robotic football team²⁰ according to a more defensive strategy on the initiative of the coach [34]; or in a bottom-up manner, with the dynamic formation (and then dissolution) of a micro-organization of type "one-two" on the initiative of some player agent [21]. Examples of abstract models of organizations are AGR [25] and MOISE+ [34].

As for services, multi-agent systems also often use various mechanisms for putting agents into relation: by

¹⁶The notion of software architectures [54] of an application focuses on the logic of the coupling between the components, independently of their internal implementation.

¹⁷Including in particular web services [18].

¹⁸In general, according to various criteria (e.g., availability, price, flexibility...).

¹⁹This mechanism of abstract role designation of the receiver will be analyzed in Section 4.2.1.

 $^{^{20}\}mathrm{As}$ in the RoboCup contest [35].

some intermediary agents, directory agents, or facilitator agents guided by the content of the message (e.g., in KQML [26]); or by some selecting and contracting mechanism, as, e.g., the contract net protocol $[57]^{21}$.

To conclude, note that the software architectures and components communities started to support automatic reconfiguration, e.g., for nomadic applications [22]. But the knowledge and social-oriented approach of multiagent 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 *quarantees* on the operability of the system.

4.2 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...;
- the mode for data transfer, e.g., unidirectional, bidirectional with value return, via a shared space...;
- the temporal coupling (in other words, the way communications are synchronized), e.g., synchronous, asynchronous, with an anticipated response (future), coordinated by a protocol...

4.2.1 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 [54]. 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. This type of mechanism²² 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 4.1).

The shared spaces (repositories) architectural style, illustrated by, e.g., blackboards and tuple-spaces (for instance, the LINDA model [31]), introduces a mode of designation of the receiver totally 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 4.1). 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 [36] (e.g., in KQML [26], to be analyzed in Section 5.2). Another type is the abstract designation of a receiver through a role, as, e.g., in the AGR (agent group role) organizational model [25]. 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 3).

Last, in certain types of multi-agent systems, in which the *environment* (physical or not)²³ is explicitly modeled, the agents can communicate via the *environment*, though inserting specific data, for example *pheromones* for ant-based algorithms. Note that, moreover, there is a current trend in multi-agent systems for promoting the environment as a *first-class abstraction*²⁴ [62].

4.2.2 Data Transfer

The mode for data transfer in object-oriented programming is bidirectional, with some return of value²⁵. It is inherited from the procedural or functional call. It corresponds (as we will see in Section 4.2.3) 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.

 $^{^{21}\}mathrm{It}$ will be discussed in Section 4.2.3 and is illustrated in Figure 5.

²²The subscription criteria and the distribution method may vary, see, e.g., the classification proposed in [23].

²³Algorithms based on *ants* and their *pheromones* can be used as a general meta-heuristic optimization method (see, e.g., [2]), the environment having then no longer relation with a physical reality.

²⁴There 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 [47].

²⁵Unless 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.

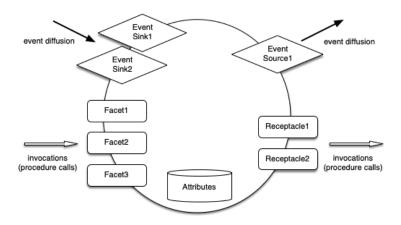


Figure 4: CCM component model

The actor model [1] introduces some unidirectional (and asynchronous, see Section 4.2.3)²⁶. 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) [63], provide the programmer with a choice between a one-way asynchronous message send and a two-way synchronous call²⁷. Component models, such as CORBA component model (CCM) [45]²⁸ 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 by event diffusion (via event sources and sinks), see Figure 4.

The *shared spaces* architectural style (see the previous paragraph Designation of the Receiver) 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 for 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 [18] (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 5.2).

Last, some possible communication via an *environment* (by adding, removing, or consuming data) represents some *indirect* mode of data transfer.

4.2.3 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 [1] 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 indicated in Section 4.2.2, some actor-based languages, such as ABCL, can provide both one-way asynchronous and two-way synchronous communication (and even other modes, such as the promise/anticipation of the response – often named future²⁹ –, which we will not develop here, see [63]). Note that Scala is an example

 $^{^{26}}$ 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.

²⁷Various types of such actor-based and object-oriented concurrent programming abstractions (action selection, activity, communication and synchronization) have been jointly modeled within the object-oriented framework Actalk [9, 10].

²⁸Note 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) [40]. However, we have chosen here to illustrate our analysis through the CCM model, for its historical and pedagogical value.

²⁹Future is a type of eager evaluation, also coined as wait by necessity [16], the exact opposite of lazy evaluation.

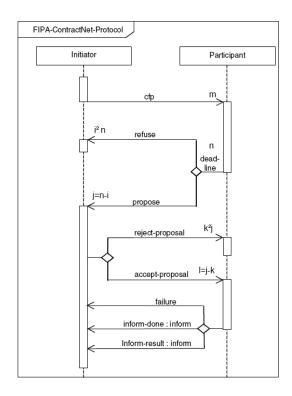


Figure 5: Contract net protocol. Figure reproduced from FIPA Contract Net Interaction Protocol Specification, Foundation for Intelligent Physical Agents, 2002.

of a programming language that integrates functional, object-oriented, and actor programming [43]. 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 [13].

Agent communication languages (ACL), in particular FIPA³⁰ ACL [27], 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...), coordination (such as a simple or iterated call for proposals – see below), negotiation, auction (e.g., English or Dutch, with an 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 5 shows the corresponding interaction diagram (as specified by FIPA [27]). Successive phases are: the broadcast of the initial call by the initiator (also named contractor), where cfp stands for call for proposals) to the participants; various proposals (or refusals) made by the participants – controlled by some deadline (timeout) for responding –; the 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 his proposal.

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 W3C³¹, but it has been replaced by the BPEL (Business Process Execution Language) and BPNM (Business Process Model and Notation) standards [41].³².

Table 2 summarizes the evolution of coupling according to the 2 main facets: structure and communication, the latter with its 3 sub-facets: designation of the receiver(s), data transfer mode, and temporal coupling (synchronization). Figure 6 illustrates it within our proposed frame of reference³³.

³⁰FIPA is the acronym for the Foundation for Intelligent Physical Agents, an IEEE Computer Society standards organization that promotes agent-based technology and the interoperability of its standards with other technologies [28].

³¹The World Wide Web Consortium standard [61].

³²We will not detail here the characteristics of services and Web services, which are the subject of standards and numerous technical specifications (see, e.g. [18] and [49], as well as [50] for an agent perspective on web services) because that would be the subject of another article.

 $^{^{33}}$ Note that the coupling flexibility evolution 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)
Commu- nication					
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) direct	Uni- directional direct	Bi- or uni- directional (events) direct	Bi- or uni- directional	Uni- directional direct or indirect (environment)
Synchro- nization	Synchronous	Asynchronous	Synchronous or asynchronous	Synchronous or asynchronous	Asynchronous or protocol

Table 2: Coupling nature

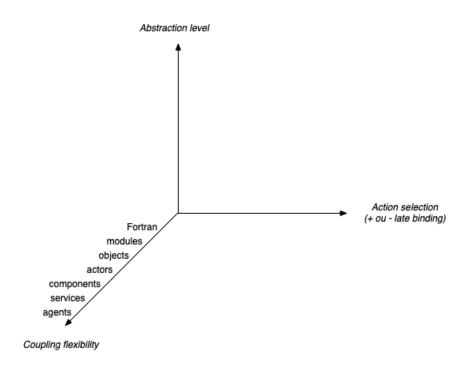


Figure 6: Coupling flexibility evolution

Abstraction Level 5

The history of programming begins with concepts very close to the machine (instructions, integers...), then progressively identifies some higher level abstractions (procedure, function, data structure, semaphore, process, object, message, component, model...). The concepts of agent and organization continue this evolution towards more abstraction as well as towards more explicit knowledge.

5.1From 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 [51]³⁴. 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., [55]), with some symbolic representation of cognitive concepts, such as: belief, qoal, desire, intention... Furthermore, such internal knowledge 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 context³⁵ for context-aware applications such as ambient intelligence [60].

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... 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...).

5.2Interoperability Languages

Let us look at interoperability middlewares, which specify and standardize the exchange of information. CORBA object-oriented middleware designed by OMG [44] 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 substituted³⁶ 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...);
- 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 [27]);
- ontology: the ontology(s) 37 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 5).

It should be noted that CORBA and ACL do not actually play exactly the same roles [58]. 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++...). 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 a 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 [26], followed by FIPA ACL [27].

 $^{^{34}\}mathrm{A}$ review of their successes, failures and prospects is proposed in [52].

³⁵[19] 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).

36 Actually more than that, as it will be explained in next paragraph.

³⁷I.e. some representation of a set of concepts, their properties and their relations.

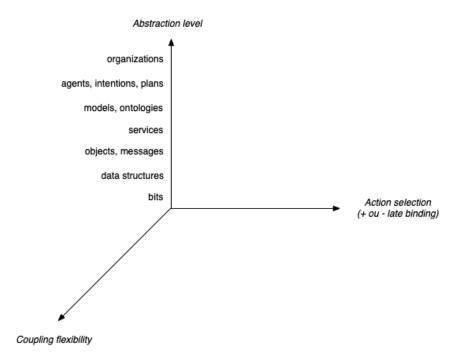


Figure 7: Abstraction level evolution

5.3 Organizational Design

It is also important to highlight the preponderant role of the design of systems 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 Cassiopée [21] precursor) 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 level³⁸.

Finally, in the evolution and the elevation of programming abstractions, as illustrated in Figure 7, we also need to mention about model driven engineering, such as model driven architecture (MDA) proposed by the OMG [46]), as a modeling level for the partial automation of the construction of applications. Note that this line of research is somehow orthogonal to a specific programming model (object-oriented, component-based, agent-oriented...). There are efforts to couple multi-agent programming and model engineering, see, e.g., [56].

6 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., [58]). They get inspiration from multiagent 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 [50]).

An important stake is therefore be 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 is to humbly contribute to clarify various programming abstractions and their respective evolution and articulation and thus

³⁸However, 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 themselves*, thus offering possibilities of *self-adaptation* and *self-organization* (see, e.g., the organizational model MOISE [34]).

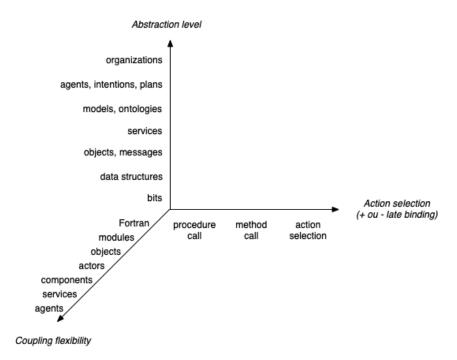


Figure 8: Programming evolution

to favor mutual awareness and possible cross-fertilization³⁹.

Acknowledgements

The premises of this study go back to an interview that we conducted with Les Gasser on the relationship between objects and agents [30], published in a special series on actors and agents [30]. We thank him for his pioneering and fundamental contribution to this reflection and we dedicate this article to his memory.

References

- [1] Agha, G.: Actors: a Model of Concurrent Computation in Distributed Systems. Series in Artificial Intelligence. MIT Press (1986)
- [2] Albert, P., Armetta, F., Hassas, S.: Agents situés: une nouvelle voie pour le développement d'applications industrielles. In: A.E. Fallah-Seghrouchni, J.P. Briot (eds.) Technologies des systèmes multi-agents et applications industrielles, IC2, pp. 101–146. Hermès/Lavoisier (2009)
- [3] Allen, R., Garlan, D.: Formal connectors. Research Report CMU-CS-94-115, Department of Computer Science, Carnegie Mellon University, Pittsburgh, PA, USA (1994)
- [4] Bachman, F., Bass, L., Buhman, C., Comella-Dorda, S., Long, F., Robert, J., Seacord, R., Wallnau, K.: Volume II: Technical concepts of component-based software engineering. Technical Report CMU/SEI-2000-TR-008 & ESC-TR-2000-007, Software Engineering Institute, Carnegie Mellon, Pittsburgh, PA, USA (2000)
- [5] Beugnard, A., Jézéquel, J.M., Plouzeau, N., Watkins, D.: Making components contract aware. IEEE Computer 32(7), 38–45 (1999)
- [6] Boissier, O.: (editor) Composants et systèmes multi-agents. L'Objet 12(4) (2006)
- [7] Bordini, R., Dastani, M., Dix, J., Seghrouchni, A.E.F.: Multi-Agent Programming: Languages, Platforms and Applications. International Book Series on Multiagent Systems, Artificial Societies and Simulated Organizations. Springer (2005)

³⁹Last section of the original paper [11] 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.

- [8] 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)
- [9] 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)
- [10] 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)
- [11] Briot, J.P.: Composants et agents : évolution de la programmation et analyse comparative. Technique et Science Informatiques (TSI) **33**(1-2), 85–115 (2014)
- [12] Briot, J.P., Demazeau, Y.: Principes et architecture des systèmes multi-agents. IC2. Hermès/Lavoisier (2001)
- [13] Briot, J.P., Guerraoui, R., Löhr, K.P.: Concurrency and distribution in object-oriented programming. Computing Surveys **30**(3), 291–329 (1998)
- [14] Briot, J.P., Meurisse, T., Peschanski, F.: Une expérience de conception et de composition de comportements d'agents à l'aide de composants. L'Objet **12**(4), 11–41 (2006). Special issue on Composants et systèmes multi-agents
- [15] 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)
- [16] Caromel, D.: Toward a method of object-oriented concurrent programming. Communications of the ACM (CACM) **36**(9), 90–102 (1993)
- [17] Castagna, G.: Covariance and contravariance: Conflict without a cause. ACM Transactions on Programming Languages and Systems (TOPLAS) 17, 431–447 (1995)
- [18] Chauvet, J.M.: Services Web avec SOAP, WSDL, UDDI, et XML. Eyrolles (2002)
- [19] 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)
- [20] 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)
- [21] 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)
- [22] 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)
- [23] Eugster, P., Felber, P., Guerraoui, R., Kermarrec, A.M.: The many faces of publish/subscribe. ACM Computing Surveys **35**(2), 114–131 (2003)
- [24] Ferber, J.: Les Systèmes Multi-Agent Vers une Intelligence Collective. InterEditions (1995)
- [25] 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)
- [26] Finin, T., Labrou, Y., Mayfield, J.: KQML as an agent communication language. In: J. Bradshaw (ed.) Software Agents, pp. 291–316. MIT-Press (1997)
- [27] FIPA: Agent Communication Language Specifications (accessed: 13/08/2021). http://www.fipa.org/repository/aclspecs.html
- [28] FIPA: Foundation for Intelligent Physical Agents (accessed: 13/08/2021). http://www.fipa.org/

- [29] 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)
- [30] Gasser, L., Briot, J.P.: Agents and concurrent objects. IEEE Concurrency 6(4), 74–81 (1998). Interview of Les Gasser by Jean-Pierre Briot
- [31] Gelernter, D., Carrierro, D.: Coordination languages and their significance. Communications of the ACM 35(2) (1992)
- [32] 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)
- [33] 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)
- [34] 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)
- [35] RoboCup Federation Inc: RoboCup (accessed: 13/08/2021). https://www.robocup.org/
- [36] Kafura, D., Briot, J.P.: Introduction to actors and agents. IEEE Concurrency 6(2), 24–29 (1998)
- [37] Lau, K.K., Wang, Z.: Software component models. IEEE Transactions on Software Engineering 33(10), 709–724 (2007)
- [38] Meyer, B.: Applying design by contract. IEEE Computer 25(10), 40–51 (1992)
- [39] Müller, J.P., Pischel, M.: The agent architecture InteRRaP: Concept and application. Technical Report RR-93-26, DFKI, Saarbrücken, Germany (1993)
- [40] 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)
- [41] 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)
- [42] Odell, J.: Objects and agents compared. Journal of Object Technology (JOT) 1(1) (2002)
- [43] Odersky, M., Spoon, L., Venners, B.: Programming in Scala. Artima (2010)
- [44] OMG: Common object request broker architecture (CORBA). Tech. rep., Object Management Group (OMG), http://www.omg.org/corba/ (accessed: 13/08/2021)
- [45] OMG: Corba component model (CCM). Tech. rep., Object Management Group (OMG), http://www.omg.org/technology/documents/formal/components.htm (accessed: 13/08/2021)
- [46] OMG: Model driven architecture (MDA). Tech. rep., Object Management Group (OMG), http://www.omg.org/mda/ (accessed: 13/08/2021)
- [47] 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)
- [48] Oussalah, M.: Ingénierie des composants Concepts, techniques et outils. Vuibert (2005)
- [49] Papazoglou, M.: Web Services & SOA, Principles and Technology, Second Edition. Pearson (2012)
- [50] Payne, T.: Web services from an agent perspective. IEEE Intelligent Systems 23(2), 12–14 (2008)
- [51] 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)
- [52] Perrot, J.F., Briot, J.P.: Introduction. L'Objet **10**(4), 11–16 (2004). Numéro spécial : Des octets aux modèles Vingt ans après, où en sont les objets ?
- [53] Peschanski, F., Meurisse, T., Briot, J.P.: Les composants logiciels : évolution technologique ou nouveau paradigme ? In: Conférence Objets, Composants, Modèles (OCM 2000), pp. 53–65. Nantes (2000)

- [54] Shaw, M., Garlan, D.: Software Architectures Perspective on an Emerging Discipline. Prentice Hall (1996)
- [55] Shoham, Y.: Agent oriented programming. Artificial Intelligence 60(1), 51–92 (1993)
- [56] 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)
- [57] 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)
- [58] 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)
- [59] Sun: Javabeans specification. Tech. rep., Sun Microsystems Inc., http://java.sun.com/products/javabeans/(2006)
- [60] Viterbo, J., Endler, M., Breitman, K., Mazuel, L., Charif, Y., Sabouret, N., Breitman, K., Seghrouchni, A.E.F., Briot, J.P.: Managing distributed and heterogeneous context for ambient intelligence. In: W. Dargie (ed.) Context-Aware Computing and Self-Managing Systems, Studies in Informatics, chap. 4, pp. 79–128. Chapman & Hall/CRC (2009)
- [61] W3C: World Wide Web Consortium (accessed: 13/08/2021). https://www.w3.org/
- [62] 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)
- [63] 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
- [64] 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)