Chapter 1

Object-oriented, component-based, agent-oriented and service-oriented paradigms in software architectures

Recent years have seen object-oriented, component-based, agent-oriented and service-oriented paradigms coexist and develop in parallel. This has led to the emergence of similar or specialist concepts that are often juxtaposed with misinterpretations of vocabulary. These misinterpretations are exacerbated by the existence of hybrid approaches that borrow elements from the four paradigms. Moreover, modern applications that combine these paradigms emphasize this ambient intertwining and the overall understanding becomes more difficult.

The purpose of this chapter is to clarify the boundaries between the paradigms by proposing a conceptual comparative framework based on two quantitative and qualitative approaches. The principle is to concentrate on differentiation of the conceptual aspects directly related to the paradigms, as opposed to an approach that compares the different technologies for implementing these paradigms. The aim is to offer architects a better understanding of the implications and consequences of choosing one or the other of these paradigms.

1.1. Introduction

According to Wikipedia “A programming paradigm is a fundamental style of computer programming that deals with how solutions to problems must be

---

Chapter written by Abdelkrim AMIRAT, Anthony HOCK-KOON and Mourad Chabane OUSSALAH.

formulated in a programming language". This chapter focuses on four key paradigms in the field of software development - namely: *Object-oriented software engineering* (OOSE), *Component-based software engineering*(CBSE), *Agent-oriented software engineering* (AOSE) and *Service-oriented software engineering*(SOSE).These paradigms will be studied and analyzed by way of construction of real-world distributed applications.

A software development paradigm specifies how an information technology solution to a problem must be formulated in accordance with clearly-defined concepts and mechanisms. It determines the order in which to deal with the problem and provides the means to develop this order, to follow its principles and to implement it in practical terms. Thus, a software development paradigm has its own particular style of developing IT solutions, in terms of analysis, design and development.

By nature, a paradigm is independent of function-specific issues; however, it can encourage certain types of application in order to support specific qualities. However, these qualities are usually associated with specific repercussions. When a paradigm is well suited to an implementation issue, it reduces the need for costly integration process and isolated solution tests by using a common conceptual framework.

In this chapter, we propose a conceptual framework based on a *top-down* approach. The principle of a *top-down* approach is to concentrate on the differentiation of conceptual aspects directly related to the paradigms, as opposed to a *bottom-up* approach that examines their technological differences. Our comparison-based conceptual framework relies on two approaches: a quantitative approach based on the concepts of product and process, and a qualitative approach based on quality criteria that organize the characteristics of each paradigm. These approaches will assist in clarifying the conceptual and technical misinterpretations of these different paradigms.

1.2. History

Figure 1.1, drawn from [SOM 04], shows the evolution of software engineering. We can see the progression from the lines of code in structured programming to current trends, or approaches such as service-oriented and model-based paradigms[1].

1. In this chapter, we will deliberately ignore the model paradigm as proposed by the OMG and focus on the object-oriented, component-based, agent oriented and services oriented paradigms, which makes for a sufficiently extensive chapter.

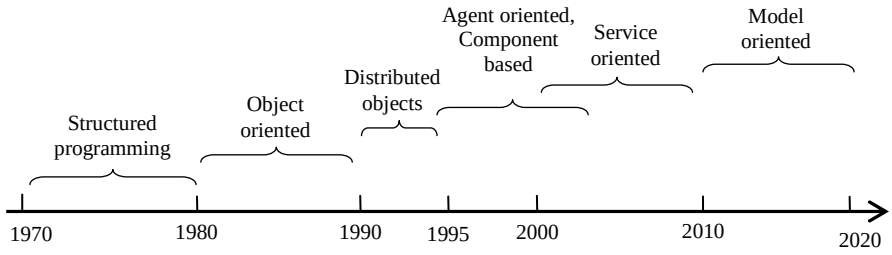

Figure 1.1. *Evolution of development paradigms*

1.2.1. *Object-oriented Paradigm*

Object-oriented Paradigm (OO) is a design-oriented and programming-oriented paradigm that emerged in the early 1960s and continued by Alan Kay's works in the 1970s [KAY 93]. It consists of the definition and interaction of software modules called objects: an object represents a concept, an idea or any entity in the physical world [OUS 99]. It has an internal structure and behavior, and it is able to communicate with other objects. The aim of OOP, therefore, is to represent these objects and their relationships: communication between objects *via* their interrelations facilitates the implementation of the intended functionalities.

The Simula-67 language lays the first foundations; those of object-oriented languages: class, polymorphism, inheritance, etc. [COX 91]. However, it was actually with Smalltalk 71,followed by Smalltalk 80 (Dan Ingalls) [GOL 83], inspired largely by Simula-67 and Lisp, that the principles of object-oriented programming, building on the work of Alan Kay's, would be established: object encapsulation, messages, typing and polymorphism (*via* sub-classification); other principles such as inheritance, are either derived from these or fall within implementation.

The 1980s witnessed the proliferation of object-oriented languages: Objective C (early 1980s), C++ (C with class structure) in 1983, Eiffel in 1984, Common Lisp Object System in 1987, etc. The 1990s saw the golden age of enhancement of object-oriented programming in different sectors of software development. Currently the object-oriented approach is considered as the reference model for other approaches.

Then, the object-oriented has been completed with the *Remote Method Invocation* (RMI) mechanism with the aim of introducing the concept of

distribution in the programming model. Remote Method Invocation is mainly based on the principle of ORB (*Object Request Broker*) [GAS 92, VIN 97].

1.2.2. *Component-based Paradigm*

The component-based paradigm was proposed by McIlroy [MCI 68] where he implemented an infrastructure on Unix using *pipeline* components and filters. Component-based development appeared in the early 1990s, in response to the failure of the object-oriented approach to meet the requirements of reuse and composition. The component-based approach extends the object-oriented paradigm by stressing the importance of reuse, the separation of problems and promotion of composition [PAP 07].

Reading and understanding an existing code is always a tedious task for developers; however, it is highly advantageous to be able to reuse an existing code in the form of a component. In fact, a developer only needs to know a component includes, and not how it was implemented. In addition, in the component-based approach, a clear distinction is made between the development of a component and that of a system. In the first case, we focus on the arrangement of the component and in the second case; we focus on the assembly and composition of compatible components.

1.2.3. *Agent-oriented Paradigm*

The agent-oriented approach appeared in the 1970s under the leadership of distributed artificial intelligence (DAI) where Hewitt [HEW 73, HEW 11] proposed the concept of actors i.e. competing interactive autonomous entities. In the mid-1990s, MAS collective models (multi-agent systems) appeared. In these models, an agent is treated as self-contained entity with certain capabilities that enable it to carry out its services or use the services of another agent through interaction. Organization-based of multi-agent systems (OMAS) are among the new models [FER 03].

Agents are distinguished by their social ability to cooperate, coordinate and negotiate with each other [HYA 96]. Autonomy and high-level interactions are the main points of difference between agent-based and object-oriented, component-based and services-based approaches. Agents can be classified into two categories:

- Reactive agents wait for an event to happen before responding to changes in their environment.
- Proactive agents take decisions on their own initiative in their environment.

Software agents have their own control *thread*, encapsulating not only their code and state, but their invocation too. These agents may also have rules and individual goals, appearing as active objects with invocation initiative. In other words, when and how an agent can act is determined by the agent itself.

In the agent model, communication is usually asynchronous. This means that there is no predefined flow of control from one agent to another. An agent can initiate an internal or external autonomous behavior at any time, and not only when it receives a message [HEW 77].

Agents can respond not only to invocations of specific methods, but also to observable events in the environment. Proactive agents can actually question the environment for events and other messages to determine what measures to take.

1.2.4. *Services-oriented Paradigm*

The service-oriented paradigm is a relatively new software development, dating from the early 2000s, and well established in the field. SOSE (*Service oriented software engineering)* is directly inspired by real-world organization methods in trade between multinationals, and is based on the classic notion of service.

The origin of service-oriented software engineering comes from requests related to systems that need to be able to withstand increasingly volatile and heterogeneous environments such as the Internet and Web services [CAS 03], ambient intelligence environments [WEI 91] or business applications run on corporate networks such as ERP[2] systems [PAP 07]. The productivity of a supplier and their responsiveness to changing needs are major issues that SOSE attempts to provide solutions to in software development.

The service is a software entity that represents a specific function. It is also an autonomous building block that does not depend on any context or external service. It is divided into operations that contain specific actions that the service can provide. A parallel can be drawn between operations and services on the one hand, and methods and classes in the OOSE on the other. SOSE also has a concept of composite service built by combining service descriptions. The implementation of service compositing takes place during the *runtime* phase.

A key element of SOSE is the pattern of interaction of services, also known as services-oriented architecture (SOA[3]) that enables a range of services to

2. ERP: *Enterprise Resource Planning*.
3. SOA: *Service Oriented Architecture*.

communicate with each other. SOA is a means for design and an understanding of a software system to provide services to applications or other services *via* the publication of tracked interfaces.

A service is an action performed (a function rendered) by a provider for a customer; however, the interaction between the supplier and customer is established *via* a mediator (which may be a bus) responsible for bringing between participants together. Services are usually implemented as coarse-grained software entities. They encompass and propose system entities. These systems can also be defined as the application layers. The concept of a service represents a processing entity that respects the following characteristics:

- Coarse grained. Operations offered by a service encapsulate several functions and operate on a wide range of data, unlike with the component-based concept.
- Interface. A service can implement several interfaces, and several services can implement a common interface.
- Architecture. Each service is described by an architecture that enables us to understand what it does, in which conditions, at what price and with which non-functional properties are involved.
- Discoverable. Before a service can be called (*bind*, *invoke*), it has to be found (*Look-up*).
- Single instance. Unlike components that are instantiated on demand and can have multiple instances at the same time, it is a single service. It corresponds to the *singleton design pattern*.
- Loosely coupled. Services are connected to customers and other services via standards. These standards ensure decoupling i.e. the reduction of dependencies. These standards are XML documents as in the case of Web services. However, several communication techniques manage the heterogeneity of services implementations so that they can still communicate. In the context of SOSE, coupling encompasses all concepts of dynamic discovery of services and automatic changing/replacement of these services.

SOSE considers an application as a set of services interacting in accordance with their roles and regardless of their location, in order to withstand heterogeneous and loosely coupled software systems. The Web service is an example of a service where we use three basic elements which are: WSDL (an XML meta-language) as a description language, UDDI registry to enable localization and a transfer protocol such as HTTP or SOAP.

Service-Oriented Architecture (SOA) is essentially a collection of services that interact and communicate with each other. This communication merely

consists of a data return or an activity (coordination of several services). Services-oriented architecture is an interaction model application that implements services. This term originated between 2000 and 2001.

There is a hierarchy of services corresponding to the different layers of the technical architecture of a solution. Services-oriented architecture is a very effective solution to the problems faced by companies in terms of reusability, interoperability and reduction of coupling between systems that implement their information systems.

SOA became mainstream with the emergence of standards such as Web services in e-commerce, B2B (Business to Business) or B2C (Business to Consumer) based on platforms like J2EE or .NET.

1.3. Software Architecture

For many years, software architecture was described in terms of boxes and lines. It was not until the early 1990s that software developers became aware of the crucial role that software architecture plays in the successful development, maintenance and evolution of their software system. A good software architecture design can lead to a product that meets customer needs and can easily be updated, whereas an improper architecture can have disastrous consequences that can lead to the withdrawal of a project [TAY 09].

1.3.1. *Object-oriented software architecture*

Object-oriented modeling creates diagrams, text specifications and programming source code based on object-oriented concepts to describe a software system. Object-oriented modeling languages are methods and techniques to analyze and represent software systems graphically. There are several methods of modeling objects such as DOSS (*Designing Object-Oriented Software*) by Wirfs-Brock, MOT (*Object-Modeling Technique*) by Rumbaugh, OOSE (*Object-Oriented Software Engineering*) by Jacobson, or OOD (*Object-Oriented Analysis and Design*) by Booch. However, nowadays, most of these methods are integrated into UML (*Unified Modeling Language*) by Booch *et al.*, and therefore, are no longer practiced by analysts. Object-oriented software architecture is used to describe a system as a collection of classes (entities to be abstracted and the encapsulation of functionalities) that can have objects (instances) and communicate between themselves by sending messages [OUS 99, OUS 05].

1.3.1.1. *Advantages and disadvantages of object-oriented software architectures*

Object-oriented software architectures offer several advantages:

- They are based on well-defined methodologies to develop systems on the basis of a set of requirements.
- They often provide direct mapping from specification to implementation.
- They are familiar with a large community of engineers and software developers.
- They are supported by commercial tools.

However, they suffer from a number of shortcomings. The most significant are:

- Significant limitations in terms of granularity and scale-up.
- Low level of object reuse partly due to the tight coupling of objects. In fact, they can communicate without using their interface.
- The structure of object-oriented applications has poor legibility (a set of files).
- Most object-oriented mechanisms are manually managed (instance creation, management of dependencies between classes, explicit method calls, etc.).
- There are few or no tools to deploy executables on different sites.
- They only specify the services provided by object implementation but do not, in any way, define the requirements of these objects.
- They provide little or no direct support to characterize and analyze non-functional properties.
- They provide a limited number of primitive interconnection mechanisms (method invocation), making it difficult to account for complex interconnections.
- They offer few solutions to facilitate the adaptation and assembly of objects.
- They find it difficult to take account of object-oriented developments (adding, deleting, modifying, changing communication methods, etc.).
- They are not suitable for building coordination patterns and complex communication.
- They have limited support for hierarchical descriptions.
- They make it difficult to define the overall systems architecture prior to the complete construction of the components.

1.3.2. *Component-based software architecture*

Component-based software architectures describe systems as a set of components (processing or storage units) that communicate with each other via connectors (interaction units). Their goals are to reduce development costs, improve the reuse of models, share common concepts between system users and finally build reusable off-the-shelf component-based heterogeneous systems. To support the development of such architectures, it is necessary to have formal notations and tools of for analyzing architectural specifications. ADL (*Architecture Description Languages*) stands as a good solution for this purpose [OUS 05, TAY 09].

1.3.2.1. *Advantages and disadvantages of component-based software architectures*

In component-based software architectures:

- Interfaces are generally first-class entities explicitly described by ports and roles.
- Interactions are separate from the calculations and are explicitly defined in most ADLs.
- Non-functional properties are taken into account.
- Hierarchical representations are semantically richer than simple inheritance relationships.
- ADLs are enhanced by architectural styles defining a design vocabulary framed by a set of constraints on this vocabulary.
- The overall description of system architecture can be specified before completing the construction of its components.
- The level of granularity of a component or connector is higher than that of an object or of an association.

However, component-based software architecture:

- Provide only high-level models, without explaining how these models can be connected to the source code. Such connections are important to preserve the integrity of the design.
- Remain an *ad hoc* concept known by the academic community. Currently, the industrial world is becoming increasingly interested in this discipline of software engineering.

- Despite the ISO/IEC/IEEE 42010:2011 standard[4], there is no real consensus because different notations and approaches for describing software architectures have been proposed.

1.3.3. *Agent-oriented software architecture*

Organization-based multi-agent systems (OMAS) are effective systems, which meet the challenges of designing large and complex Multi-Agent Systems (MAS). Multi-Agent Systems is a paradigm for understanding and building distributed systems, where it is assumed that the processing elements - i.e., agents, which are autonomous entities able to communicate - have a partial knowledge of what surrounds them and have their own particular behavior, as well as a capacity to execute themselves independently (see Figure 1.2). An agent acting on behalf of a third party (another agent, a user) that it represents without necessarily being connected to it, reacts and interacts with other agents. The social capacity for cooperation, coordination and negotiation between agents is one of their main characteristics [WOO 09].

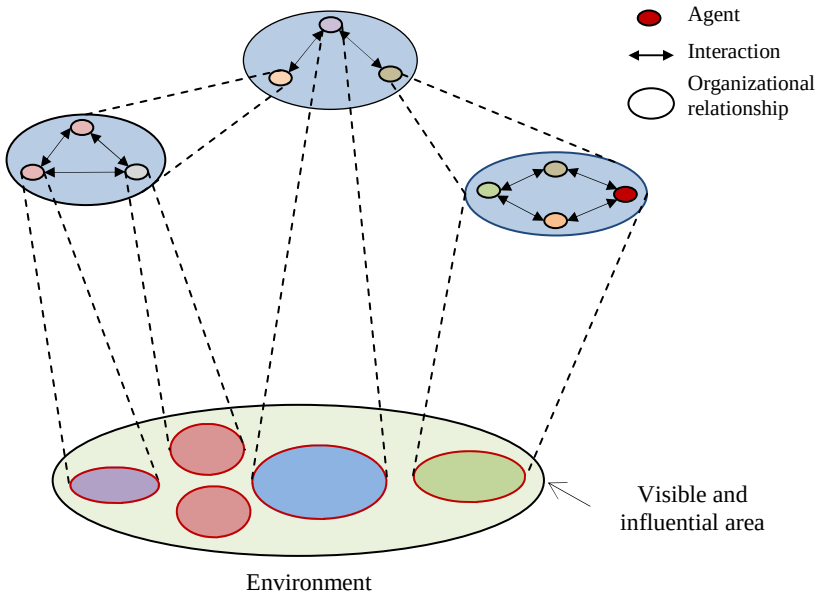

Figure 1.2. *Canonical view of organizational multi-agent system [JEN 01]*

4. www.iso-architecture.org/ieee-1471/.

To summarize, a framework for specifying agents must be able to capture at least the following aspects of a multi-agent system:

- Beliefs that the agents have.
- Interactions that agents have with their environment.
- The objectives that officials are trying to achieve.
- Actions that agents perform and the impact of these actions.

1.3.3.1. *Advantages and disadvantages of agent-oriented software architecture*

In agent-oriented programming the concept of software architecture is replaced by a further knowledge-driven concept called organization. An organization is made up of a set of roles and relationships between these roles. Figure 1.3 shows that a role can be played by one or more agents and an agent could also potentially play more than one role simultaneously. A role is an abstraction of an agent; it allows for a more generic description of the architecture as well as the interaction between agents [WOO 09].

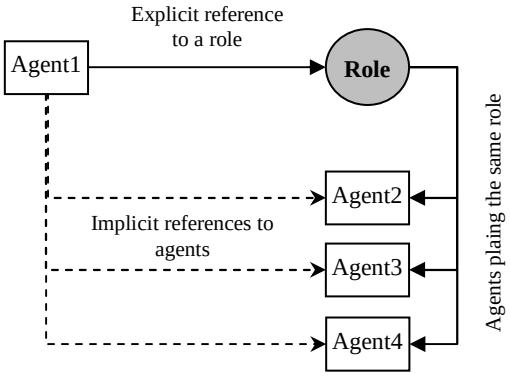

Figure 1.3. Coupling between agents

Generally:

- Agents-oriented architectures support competition and distribution.
- Agent-oriented architectures integrate business and technical perspectives.
- Relationships between agents are therefore very dynamic and partly managed independently or *via* organizations.
- Multi-agent systems take the coupling and collaboration concept between entities further (coordination, decomposition, negotiation, etc.).
- MASs use coupling mechanisms dynamically and indirectly (intermediary agent, directory agent, etc.).

- MASs propose semantic coupling guided by knowledge and by a social organization of work.

By contrast, in agent-oriented software architecture:

- The agent-oriented paradigm does not support non-functional properties.
- The usually have only one input, thus they are not compositional.
- Agent-oriented architectures are generally difficult to verify.

1.3.4. *Services oriented architecture (SOA)*

SOSE is based on the concept of service oriented architecture (SOA [OAS 08, PAP 07]) which defines a conceptual framework to organize the construction of application based on services. SOA introduces the concepts of service providers and consumers.

- A service provider is the actor responsible for the development, deployment, execution and maintenance of the service when it is required. In addition, when the service expires the provider takes care of the termination of the service activities.
- A service consumer is the actor who uses services according to their needs.

In the beginning, suppliers and consumers are independent - i.e., the supplier during the implementation of its services, has no prior knowledge about the future consumers, nor how they might reuse that service. Thus, the SOA is based on a third actor called the *service broker* [OAS 08].

The *service broker* is the actor associated with a service registry that enables the relationship between consumers and suppliers who are unaware of each other. Suppliers publish their services in these registries, which are then used by consumers to identify those that match their needs.

Suppliers and consumers commit to a contract of use, in terms of respect for the service interface for the consumer and compliance with functional and non-functional properties promised to the supplier. Figure 1.4 summarizes the organization of a services-oriented architecture.

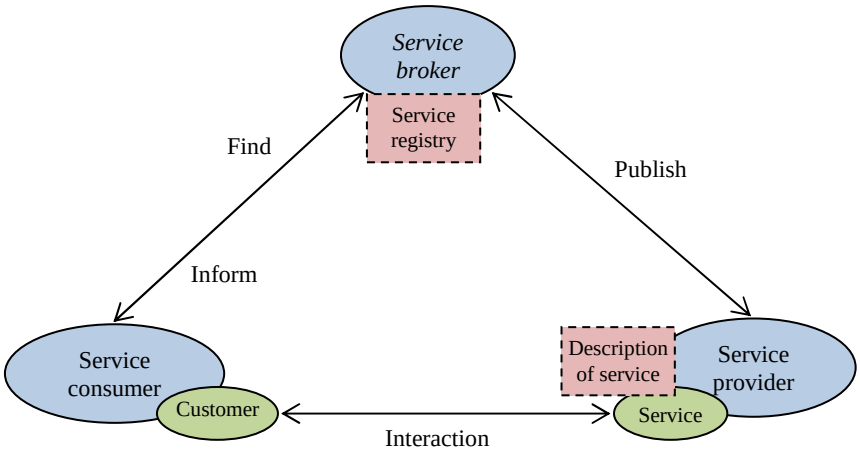

Figure 1.4.Services-oriented architecture organization

1.3.4.1. *Advantages and disadvantages of services-oriented software architecture*

- SOA provides dynamicity *via* the mechanism of discovery and dynamic service selection.
- The service-oriented paradigm supports distribution as well as the management of non-functional properties.
- The service-oriented paradigm does not support the aspect of competition (in the sense of having parallel processing capabilities).
- SOA offers high internal consistency (using a pivot exchange format, usually XML).
- SOA provides loose external coupling (using an interoperable interface layer, usually a Web service, and through the discovery and dynamic selection of services).
- SOA can develop a complex product by integrating different products from different suppliers, regardless of the platform and technology used. Thus, it helps to manage the complexity involved.

However, service-oriented software architectures:

- Are not suitable for applications with GUI functionalities. These applications could become more complex if they use SOA architectures that require a large volume of data exchange.

- Also in the case of standalone application or for short term usage, the SOA will become a burden.
- Performance problem of SOA, complex mechanism, too many exchanged messages, complexity overkill for a number of software packages, not suitable for systems with very strong time constraints, etc.

1.4. The two dimensions of the conceptual framework for comparison: quantitative and qualitative

The aim of our conceptual framework for comparison is to fill the gaps around the clear identification of differences between OOSE, CBSE, AOSE and SOSE. The goal is to provide a better understanding to users by a comparative summary of the four paradigms in order to assist them in deciding on the use of one or the other of these paradigms. This aspiration involves a grasp of their respective concepts, in their definition and then in the analysis of impacts on quality.

This comparison between object, component, agent and services serves the same effort and the same purpose as the comparison between objects and components [OUS 05, SZY 02]. The common goal is the analysis and understanding of the differences in a unique comprehensible framework.

Thus, the approach we develop follows a *top-down* pattern, which as opposed to previous *bottom-up* works, focuses initially on the conceptual levels, directly on the paradigms before seeking to derive the qualitative implications. This high level focus allows the definition of an overall framework capable of handling four paradigms. In this definition of the comparative framework, we seek both:

- Generality in identifying categories and sub-categories of the comparison framework that should not be dependent on a particular paradigm, but rather provide an outside perspective on which elements of the four paradigms may be projected. This generality enables us not to favor one paradigm over another, and also ensures the reusability of the framework, which can be used to compare various other development paradigms.
- Minimalism in the selected categories and classified elements, which must only extract the essence of the paradigms required to identify their differences.
- Completeness in identifying differences that allows us to fully understand the impact on the quality of the choice of one paradigm over another. Completeness of this framework gives the opportunity for users to customize the qualitative analysis.

1.4.1. *Conceptual differences*

The four paradigms studied have a very similar approach based on the construction of systems from existing or future software entities. They have a common goal of maximizing reusability that is directly derived from the object. They share the same overall development process that consists of identifying software entities (object, agent, component or services) that meet the needs, and then combining these entities to make the final application. They are based on the same concepts of composition, for the construction of new entities from existing ones to ensure a consistent approach where any entity can be seen as an object, agent, component or service. Thus, this approach facilitates incremental development and exploitation of knowledge.

However, although these four paradigms have the same overall goal, the concepts behind the notions of objects, components, agents, and services are different.

Thus, we confront the following four aspects:

- Difference in utilization and owner's responsibility.
- Difference in coupling.
- Difference in granularity.
- Difference in cooperation and problem-solving.

1.4.1.1. *Difference in use and owner's responsibility*

A component is called "off the shelf" [CRN 06, HEI 01] by adopting a piece of technology, the component, which is available for developers. The latter recover a block of software component and ensure its incorporation based on their requirements.

A service focuses on the use of a function provided by a third party [DUS 05, NIT 08, OAS 08, THE 08]. A service consumer only uses the result from the invocation of the target service.

These two views seem close at first; however, they have a significant impact on the allocation of responsibilities between supplier and consumer. To illustrate this distinction, we take an example of the video game industry on PC. This industry is mainly based on two models of content distribution:

- Classical model: purchasing a game in a specialist shop or downloading on the Internet.
- Cloud gaming model: purchasing a subscription to play available games directly on an Internet platform.

The classic model illustrates the object-oriented, component-based and agent-oriented approach. The said cloud gaming model illustrates the service-oriented approach.

14.1.1.1. Responsibility of an object, component or agent

The first classic model corresponds to a player who buys a copy of his game. This copy is collected either on a physical medium, usually a DVD or in a dematerialized form (cloud) via download platforms such as STEAM[5]. The player is then responsible for installing the game on his own machine, i.e. its deployment. It is only after this installation that he can launch the application and start playing.

This distribution model corresponds to a component-based approach. Typically, the game (the component) comes with an instruction manual (the documentation) that defines a number of consumer-end constraints. These constraints are of two kinds:

- Deployment constraints: the provider of a PC video game sets the minimum system requirements in terms of computing power (CPU, graphics card, RAM, etc.), storage capacity (hard drive), audio resources, etc. The customer's system must meet these requirements to be able to install and run the game. The installation process itself presents constraints whether it is the exact location on the hard disk or the connections requirements to the Internet, key authentication, etc. In OOSE, CBSE and AOSE these installations constraints are typically defined by the chosen component model [CRN 11]. Each model is associated with a particular system environment before it can be used. Moreover, this model provides deployment rules associated to these components.
- Usage constraints: each game provides a list of specific commands that determines how to interact with it and the resulting actions that are necessary to progress through the levels (Game play, Level design, etc.).

These elements provide the rules to be complied with if the user wants to take full advantage of the proposed experiment. In OOSE, CBSE and AOSE these user constraints are typically defined by the contractual interface of the entity (object, component and agent). Compliance with this interface is crucial to ensure the correct use of resources according to the possibilities previously determined by the supplier of the entity.

5. http://store.streampowered.com.

1.4.1.1.2. Responsibility of a service

The second distribution model, called *Cloud gaming*, illustrates the concept of service-oriented. In this model, the player pays the right to play a game that is running on a remote platform under the responsibility of the supplier. He only needs the interface and the appropriate connection to access the platform. In fact, the player is no longer responsible for operating the game on his own machine. The only information he requires is how to access this platform and how to play the game. Hence, deployment constraints no longer exist in relation to the installation of the game; only usage constraints remain. This lack of operation has several advantages. On the one hand, it simplifies the exploitation of resources by removing efforts that accompany the understanding of the installation phases. On the other hand, it ensures the optimal use of these resources. In fact, the application runs directly on the provider's environment. The latter therefore has full control of its execution. Thus, it is more likely to ensure the quality promised to its customers.

In our example, the quality of a video game (fluidity, graphics, etc.) varies depending on the system on which it is installed. Being run on a remote platform, this game has the same quality for each player connected. In addition, users who originally did not have the required system configurations will benefit from this service. Thus, constraints on the customer only decrease to their communication capacity.

Finally, another significant advantage of this service-oriented model relationship between customers and suppliers is the transparency of service developments as long as the latter do not change the initial usage constraints (connection interface, protocols, etc.). As it is, the new versions are directly accessible without the need to adapt on the consumer-end. On the contrary, in a component-based approach, if the customer wants to take advantage of these developments he must collect and deploy the game himself. Problems associated with this deployment may occur if the customer's system no longer supports the updated component. Cloud gaming illustrates this advantage where different versions of the same game follow one from the other in a transparent manner to users. As for the classic distribution, it requires players to collect a particular patch and then its deployment on their machine in order to develop the version of the game. These new versions can potentially require a hardware upgrade at the consumer-end (for example, to support an improved graphics engine) whereas it is not required in the Cloud gaming. Thus, collecting the patch, its installation and the ability to use the new version of the game may incur additional costs. These additional costs are generally not present in the service-oriented approach where the customer pays for this function whereas in the component-based

approach the customer pays for the component at a time and within a given release version.

However, the main drawback of this service-oriented relationship between the customer and the service provider is the total reliance of the first system to the second system as well as the reliance on different media of communication between them. As it is, a failure of these elements which are outside the sphere of the customer's actions sees its inability to act on the issue. In return, it is the contract previously established with the supplier that characterizes the consequences of these failures in terms of compensation for the customer.

Within the framework of *Cloud gaming*, these failures, which are out of the customer's control, are, for example, an error in the game's platform or even loss of Internet connectivity linked to the ISP. Thus, service-oriented paradigm pushes the owner's responsibility to the maximum compared to the component-based paradigm and therefore decreases the customer's responsibility. Indeed, the CBSE, the *off-the-shelf* approach, implies that the supplier is solely responsible for the development of its component, the associated quality of service required and its maintenance.

In the SOSE approach, the supplier is also responsible for the deployment, execution and management of their service. The service consumer is solely responsible for the communication and for compliance of the usage constraints.

1.4.1.1.3. Multitenant Nature

An application is called "multitenant" [JAC 05] if it offers functionalities to many users simultaneously. It therefore manages numerous instances at the same time and allow for hosting multiple isolated instances in order to guarantee accurate results to its various customers.

Similarly, an instance being run is dedicated to manage multiple parallel connections. In our example of video games, Cloud Gaming platforms support a large number of players in parallel. For each of these players, they must maintain a particular context in order to retain their respective information. This information is of two kinds:

- *Contract* groups the set of data related to the contract between the customer and the supplier that govern the use of the service (in our example: monthly subscription account number, quality, etc.).
- *Runtime* groups the set of data required to run the application throughout the use of the service (in our example: experience gained, games played,

persistent universes, etc., in order to reproduce exactly the status where the player stopped in his game).

This multitenant principle is not necessary for an object/component/agent. In fact, although it may belong to multiple compositions, at runtime, different instances of the component are created and each are created under the responsibility of a customer in the context of a particular composition.

To conclude, from a usage and owner's responsibility point of view, object-oriented, component-based and agent-oriented paradigms are close.

1.4.1.2. *Difference relating to coupling*

Coupling is a concept that we identify as one of the key breaking points between OOSE/CBSE and AOSE/SOSE. This concept expresses all possible dependencies between conceptual and software entities. Reducing coupling guarantees a number of intuitive benefits in terms of isolating errors, easing additions and removal of entities reused, reconfiguration, etc.

In fact, OOSE and CBSE have a broad mandate in the type of applications they wish to implement, whereas the SOSE and AOSE mechanisms are built to support the development of applications that run on highly volatile, cooperative and heterogeneous environments.

This difference is consecutively illustrated by their respective connection for, on the one hand, the management of heterogeneities, and, on the other hand, of the automation of other mechanisms.

1.4.1.2.1. Management of heterogeneities

The aim of the service-oriented paradigm is the independence it has with implementation technologies. A service must be accessible and usable without any assumption on its implementation, on the potential users or on how to use this service. This problem is well known in CBSE but is not as critical as in the SOSE issue. As it is, there are a large number of component models [CRN 11]. To develop a new system, the designer must choose a particular model and use only the components complying with this model as the collaboration between different models is very difficult [CRN 06]. Thus, although the CBSE has proven its effectiveness in software reuse and maintainability, it does not specifically target certain difficulties encountered by developers in relation to changes in platforms, protocols, devices, Internet, etc. [BRE 07].

For its part, the SOSE advocates a single homogeneous service-oriented model [ERI 08], to be standardized and used by all, to encapsulate all types of resources and hide their heterogeneous nature during development.

1.4.1.2.2. Comparison to on automating mechanisms

Automation contributes to the definition of the SOSE itself, and therefore the vast majority of research seeks to automate their mechanisms such as service publication, discovery, selection, composition, etc. As it is the decoupling between requirements and services used, discovery at runtime, the definition of collaboration and finally dynamic establishment of communications were the main goals set from the start in the development of the service paradigm. This principle of automation is pushed to its maximum by the concept of self-adaptation [NIT 08], which seeks to coordinate all mechanisms related to the service-oriented paradigm to allow for reactive or proactive contextual adaptations.

Although the process automation is a key element of research in CBSE and represents many of its current challenges, it is not an integral part of the conceptual origin of the CBSE or the definition of a component model [CRN 11].

Thus, with respect to the heterogeneity and automation, SOSE aims to produce loose coupling at all levels, from development to execution.

1.4.1.3. Difference in granularity

In the field of software engineering, granularity corresponds to a relative measurement of the size of architectural elements that make up the applications. The software engineering community then spoke of coarse-grained systems or fine-grained systems [BEI 07], which are respectively associated by their composition by assembly of software blocks of high granularity and low granularity. These notions of high and low granularity are determined by the importance of the encapsulated resources by architectural elements. This importance is relative to the underlying complexity of the implementation and usage of these resources.

Understanding granularity became prominent with the development of CBSE [BEI 07, MOH 08]. As it is, the granularity represents one of the first distinctive points between an object and a component. The object responds to the lack of clarity, understanding and thus the handling of systems which are decomposed into too many objects or too large objects. Thus, different component models offer different granularities [BEI 07], and these proposed varieties for the size of

the building blocks reinforce the importance of choice in the decoupling of the application in order to maximize the quality of the resulting architecture.

The concept of granularity is intuitively understandable, which counteracts with the vagueness of its formalization where the clear delineation between high granularity and low granularity remains to be defined. However, the current understanding is sufficient to establish a hierarchy between SOSE, CBSE, AOSE and OOSE, where service-oriented paradigms are usually described more coarse-grained than component-based paradigms, in the same way that the component-based paradigms are typically seen as coarse-grained in relation to agent-oriented paradigms and fine-grained in relation to object-oriented paradigms.

We justify this comparison of granularity between the service-oriented/component-based/agent-oriented/object-oriented with two commonly encountered realities:

- Technical reality: where component-based models are often used to build new SOSE services from scratch or from legacy systems. CBSE related technologies can intervene at all phases for SOSE system realization from the services implementation to their adaptations in order to integrate them taking care of the heterogeneities (as different runtime environments, languages, protocols, interfaces, etc.) or even in order to provide the level of abstraction necessary for the composition of pre-existing services. This relationship between service-oriented paradigms and component-based paradigms is the same between component-based/agent-oriented and object-oriented paradigms, where the object-oriented paradigms are commonly used to implement components or agents.
- Conceptual reality: linked to the very nature of the service and processes associated with it. The previous sections have highlighted a set of inherent properties of the SOSE such as loose coupling, heterogeneity management, automation degree, the distribution of responsibilities or even the multitenant. Although these concepts are already present in the CBSE, the thrust of SOSE is to push them to their maximum. To ensure these developments, complex processes must be executed. Thus, the coarse-grained only nature of the service-oriented paradigm comes from a need for balance or dilemma between the cost of support of the service processes, the size of the encapsulated resources and the relevance of their placement on the network.

The technical reality is offset against the component-based approaches such as [AND 08, OAS 09] which, during the implementation of the SOSE applications consider the service of the interface of a component as a service within SOSE term. In this respect, the SOSE service is seen as a subset of the

interface. However, the ratio of the implementation of one by the other remains the same.

1.4.1.4. Difference of cooperation and problem-solving

The concept of cooperation and problem-solving is a concept stemming from the field of distributed artificial intelligence (DAI) in coordination with the multi-agent approach. The main problem in the study of cooperation in distributed problem-solving is to understand how agents wishing to accommodate each other, may interact with each other to form an effective team. Two forms of cooperation are defined, the sharing of tasks and sharing of results, which correspond generally to discerned phases in the study of problem-solving. In both of these types of cooperation, R. Davis and R. Smith are particularly interested in its control and communication. In the division of tasks, the control is directed by the goals and the agents are represented by the tasks they are committed to perform; the problem lies in the distribution of the tasks. In sharing results, the control is data-driven, the agents are represented by knowledge resources, and the problem lies in the communication of the results [BOU 92, SMI 81].

A cooperative strategy is necessary to perform tasks effectively whose problem-solving involves several agents. The purpose of a strategy is to ensure overall consistency from local decisions and enable the effective use of communication. Two classes of cooperative strategies are defined: organizational strategies and the distribution of information strategies. The first class deals with the decomposition of a global task into subtasks and assigning these subtasks to the agents. They aim to identify the most appropriate agent to decide which plan to follow. For instance, an organizational strategy chooses an agent, which has the largest selection of possible actions. Strategies on the distribution of information indicate how and when agents must communicate. For example, one of these strategies specifies that we should not repeatedly send the same information to an agent [BOU 92, CAM 83].

Cooperation refers to a judgment value on the overall activity of a set of agents. The judgment of cooperation is influenced by several indicators such as the number and the persistence of conflicts as well as the synchronization of actions of different agents. The mechanisms which allow us to weigh these indicators are called cooperation processes.

Edmund H. Durfee has identified the cooperation indicators. These indicators were empirically determined from the observation of cooperative situations. The following list of indicators is not exhaustive [DUR 89]:

- Coordination of actions, this indicator relates to the adjustment of the direction of the agents' actions over time (synchronization) and in space.
- Parallelization, this indicator is based on the distribution of tasks and their concurrent execution.
- The sharing of resources, this indicator relates to the use of resources and skills such as information, results and equipment.
- Robustness, this indicator relates to the ability of the system to compensate for the failure of an agent.
- Non-redundancy, this indicator reflects the lack of redundant activities, for instance, selective communication.
- The non-persistence of conflict, this indicator reflects the lack of blocking situations; it is based on the ability of agents to prevent conflicts or to solve them by default.

The cooperation and problem-solving concept is absent in the OOSE, the CBSE and the SOSE because their basic entities are reactive and not proactive nature as it is in the case of the agents.

1.4.1.5. *Summary of conceptual differences*

We have shown our conceptual framework for comparison between the four paradigms. The first purpose is to complete the continuing lack not covered by the literature about the clear specification of the conceptual differences between object-oriented, component-based, agent-oriented and service-oriented paradigms.

We therefore chose a top-down approach, which focuses firstly, on the conceptual aspects of the different paradigms before developing qualities, which are derived from them.

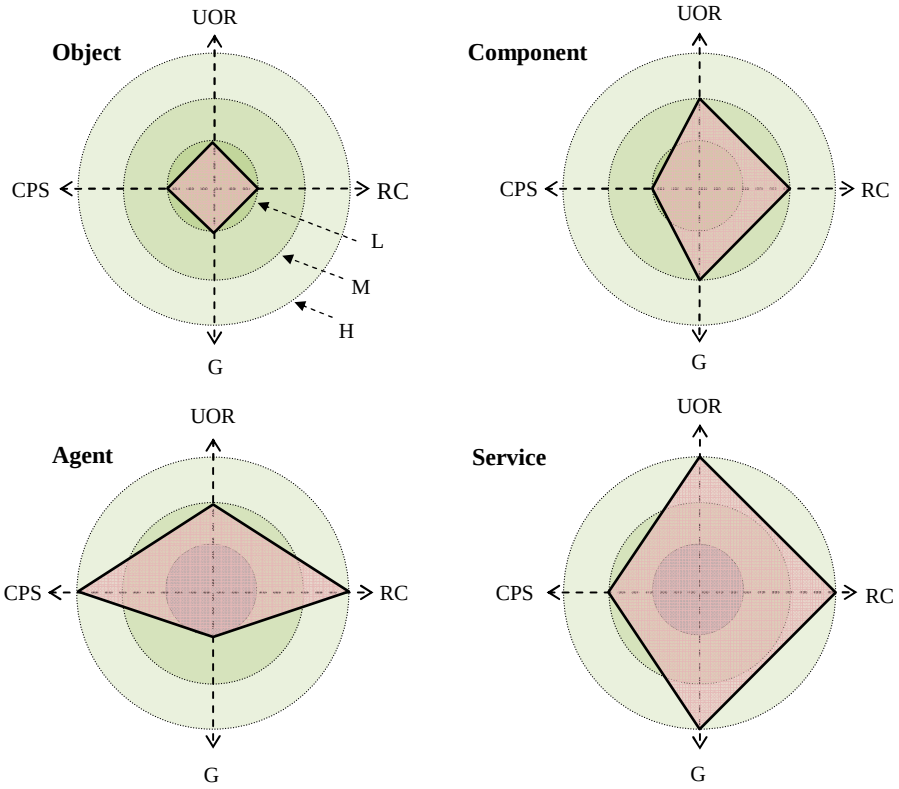

Figure 1.5. *Summary of conceptual differences between the four paradigms (UOR: Usage and Owner's Responsibility; RC: Relation to Coupling; G: Granularity; CRP: Cooperation and Problem-Solving; L: Low; M: Medium; H: High)*

1.4.2. *Quantitative dimension*

Structural elements and mechanisms, which characterize the four paradigms, can be classified into two categories: products and processes.

1.4.2.1. Product and process

A product is a software or conceptual entity that is the result of an action or process. A process is an action or series of actions that is used to create or

modify a product and thus *obtain* a product as a result. Products are divided into two subcategories:

- Simple architectural elements: the basic building blocks of a paradigm;
- Composite architectural elements: complex products built from existing architectural elements. Their structure clearly identifies the reused architectural elements and their relationships.

Each sub-category is further divided into two groups according to two levels of abstraction: the *design-time* and *runtime*.

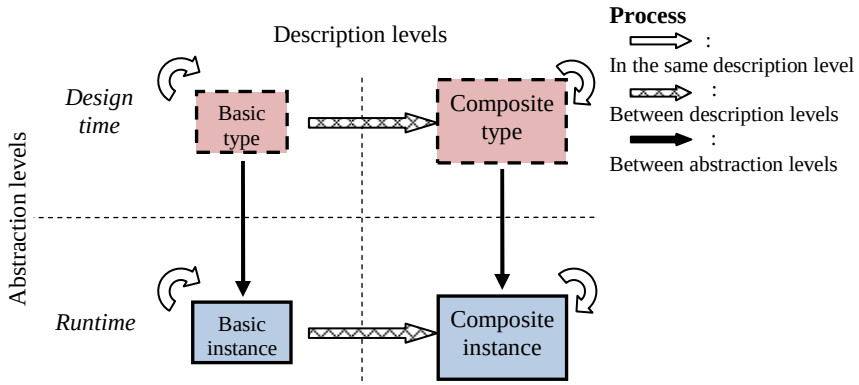

Figure 1.6. *Abstraction and description levels: distribution of products and processes*

The process category focuses on the principle of reuse, i.e. how to reuse software entities in order to build new composite ones. Conventionally, a component can be a single or composite architectural element. These notions of component and composite define two levels of description. Thus, processes are grouped according to the levels of abstraction and description:

- In the same level of description: this category groups processes that target and generate products of the same level of description (Figure 1.6: white arrows). This category is divided into both design-time and runtime.
- Between levels of description: this category groups processes that target products of two different levels of description (Figure 1.6: dashed arrows).
- Between levels of abstraction: This category represents the processes that ensure the transformation of products from the design-time to runtime (see Figure 1.6: black arrows).

Figure 1.6 shows the distribution of products and processes on a single representation. A composite "A" is made up of a set of components such as "B" which is defined as a simple architectural element. Both products have their performances at design-time and runtime. The various arrows represent processes that are being studied. The white arrows are the processes related to the same level of description and abstraction. The dashed arrows are the processes that make the link between the levels of description and have their representation at the design-time and the runtime. The black arrows are the processes that make the link between the levels of abstraction and thus ensure the transformation from the design-time to runtime.

1.4.2.2. *Comparison between paradigms*

1.4.2.2.1. Product

Single Architectural Elements

Single architectural elements of the object-oriented paradigms are the *class* at design-time and the instance (object) at runtime. The same distinction is made for the CBSE between component type and connector type products [AMI 09, GAR 97] and their component and connector instances.

Connectors [CRN 11] are mediators in connections between components, i.e. they are used as intermediaries between these constituent components. They have a two-fold function: (i) enabling indirect compositions between components and (ii) the introduction of additional functions through the glue code, which they encapsulate.

In AOSE, we describe an entity that is capable of acting in an environment that can communicate directly with other agents as a "single agent"; it has its own resources and skills and provides services to its environment. The concept of the single agent is used interchangeably in the design-time and runtime phases.

In SOSE, the boundary between levels of abstraction is much less clear and most existing work refers to a service as a runtime entity [STO 05, THE 08]. However, a notion of abstract service exists in some approaches [CAV 09]. This concept is used to distinguish between the requirements sought by the architect to define its application and services actually available in the system to meet these requirements. However, an exact clarification between abstract service and concrete service remains to be defined. We also mention the concept of service description, which is a major product of the SOSE [OAS 08]. As it is, each runtime representation of a service is associated with its service description, which is the target of many processes involved in the exploitation of resources.

Composite Architectural Elements

The four paradigms share the notion of composite. The object-oriented paradigm is based on the concepts of composite class and composite object. The CBSE relies on concepts of configuration and composite component types at the design-time. For their runtime, it relies on their configuration and composite component instances.

In the context of this study we consider a composite agent as a multi-agent system which is composed of a set of single agents representing active entities of the system with a set of relations that unite the agents between themselves. However, neither the agents nor the MAS are explicitly composable in contrast to the Organization-based agent systems which are compositional.

The notion of service composition and, ultimately, of composite service of the SOSE is mainly at the runtime. Indeed, most of the existing works consider the composite service as the execution of a collaborative scheme between services by a composition engine. However, some approaches [GEE 08, ZEN 03] introduce instantiation concepts of a collaborative scheme from abstract templates that describe them. We choose to consider this similar representation with OO types of collaborative schemes such as design-time entities, and instances of collaborative schemes such as runtime entities. In addition, a collaborative scheme is classically associated with two patterns of coordination of services, such as choreography and orchestration [RSA 08], which have technologies that support their representation in design-time and runtime. We define a composite service encapsulating a composition of services in a similar manner in the composite service type and composite service instance.

1.4.2.2.2. Process

In order to elaborate the main differences between paradigms, we describe a selection of the most relevant and widely accepted processes by the community.

At the same level of description:

Design-time. The object-oriented paradigm is primarily based on the process of association and inheritance. The CBSE is based on the horizontal composition [BAR 06, CRN 11] between architectural elements of the same level of description. This horizontal composition corresponds to the process of establishing connections between components. We can also mention versioning, selective inheritance and refinement processes. In the same level of description, the SOSE processes focus mainly on handling collaborative schemes between

services. We mention the process of choreography which is one of the principle supports for reuse and does express direct communications between services;

Runtime. Communication processes between architectural elements are the major concern in this category. OO and CBSE are based mainly on call function processes, while the SOSE inevitably includes additional processes. Typically, services have to be discovered and selected dynamically (process discovery and selection of services). Then these services coordinate their actions through a process of choreography that defines the succession of invocations of service. In addition, a front-end process of service publication is required to make the service available to potential customers (see Figure 1.4).

Between levels of description:

Design-time. The OO is based on the composition process to produce composite components. The CBSE is based on vertical composition that links components and composites. Vertical composition (or hierarchical) [BAR 06, CRN 11] consists of a sub-component encapsulated in a composite component. This composition is anti-reflexive to avoid cycles, i.e. that the same component cannot be found at several levels of the hierarchy. It assumes the consistent combinations of behavior for the composite with the behavior of its constituents. Moreover, the constituents are hidden for the requests of the composite customers. The SOSE is based on the orchestration process that models vertical communications between the composite service and its constituent services;

Runtime. The communication processes between constituents and composites are the essence of this category. OO and CBSE are based on different call processes. In CBSE, these calls are sometimes referred to as the process of delegation. In SOSE, the coordination of the process of services invocations from the composite towards its constituents is called orchestration. Similarly, the process of discovery and dynamic selection of services are required to identify the constituents of the composite service.

Between levels of abstraction:

The OO and the CBSE are based on the instantiation process to link types to their instances. The AOSE is based on the concepts of generic role (part of the *design-time*) and specific role to a domain (part of the *runtime*). The SOSE is based on the concepts of abstract service and concrete service respectively as elements of *design-time* and *runtime* [HOC 11]. However, the transition from one to the other is based on the discovery process and selection of services process. The transition from one type of collaborative scheme to an instance of collaborative scheme is based on instantiation. The transition from one type of

composite service to an instance of composite service corresponds to the combination of discovery and selection of its constituent services and the instantiation of collaborative schemes, which guide their behavior.

Table 1.1 shows a summary of the comparative study between paradigms from the product and process point of view.

Product		**Object**	**Component**	**Agent**	**Service**
Single Element	*Design-time*	Class	Component type, Connector type	Agent	Abstract service
	Runtime	Object	Component, Connector	Specific role to a domain	Concrete service, Description of service
Composite Element	*Design-time*	Composite class	Type of configuration, Type of composite component	SMA / Organization	Type of collaborative scheme, Type of composite service
	Runtime	Composite Object	Configuration, Composite component	Composite role	Instance of collaborative scheme, Instance of composite service

Process		**Object**	**Component**	**Agent**	**Service**
In the same level of abstraction	*Design-time*	Association, Inheritance	Horizontal Composition, Selective inheritance, Versioning, Refinement	Multiple roles	Choreography
	Runtime	Call method	Call function	Call transmission	Choreography, Discovery and selection, Invocation, Publication
Between levels of description	*Design-time*	Composition	Vertical Composition	Composition of roles	Orchestration
	Runtime	Call method	Call function; Delegation	Call transmission	Orchestration, Invocation, Discovery and selection
Between levels of abstraction		Instantiation	Instantiation	Specific role to a domain + Individual knowledge	Discovery and selection, Instantiation of scheme

Table 1.1. *Product and process: comparison between paradigms*

1.4.3. *Qualitative dimension*

Existing research studies related to software quality define a number of criteria such as performance, safety, robustness, flexibility, development, etc. [BIA 07, KIT 96]. Each of these studies has outlined their own organization of these criteria. The definition of these quality criteria and the way to apprehend them are being shaped based on the perspective of the target user via these measurement frameworks. Indeed, understanding a quality can vary between the stakeholders involved; whether they are architects, designers, developers or others. In addition, the scope of the system directly influences the importance of these criteria. We therefore try to cover all of these variations by offering the ability for users to define their own vision of the qualities that interest them most. In the first instance, we identify the set of factors of these paradigms that influence software quality. Then we compare OOSE, CBSE, SOSE and AOSE approaches following these factors. Secondly, the user defines the quality criteria that they want to measure by combining the previous results.

Using the various previous analyzes and by placing the four object-oriented, component-based, agent-oriented and service-oriented paradigms within the conceptual framework, it emerges that they share the following quality factors in common:

- *Reusability*: support and easiness of a product or a process related to a software development paradigm to be reused in the same way or through a number of changes.
- *Composability:* support and ease of a software development paradigm to safely combine architectural elements to construct new systems or composite architectural elements.
- *Dynamicity:* support and ease of a paradigm to develop applications that can adapt their behavior dynamically, automatically and autonomously to meet changing requirements and changing contexts as well as the possibilities of errors.

These three factors represent the qualitative nature that led to the definition of the object-oriented, component-based, service-oriented and agent-oriented development paradigms. Figure 1.7 illustrates this analysis and provides a high-level view of their primary points of interest and traces the chronological evolution of the concerns for the software engineering community.

Reusability is the oldest of the three concerns. The earlier developers quickly became aware of code repetition in an application and have therefore sought to define mechanisms to limit repetition. The object-oriented paradigm focuses on

this concern and its development is one of the outcomes of this research. The object-oriented concept facilitates the conservation and the transfer of experience gained across different systems. It further deepens reuse, which, at the outset it was intended to reuse the code as it is through the inheritance process that helps to evolve saved data and behavior in order to meet special requirements. Thus, the object-oriented paradigm provides high reusability which paved the way for applications to more complex applications and thus to the identification of new limits in terms of granularity, of software architecture, communication abstraction, etc. These limits have therefore led to a shift of concern to composability.

Thus, the software engineering community has developed and introduced the CBSE to overcome this new challenge. The famous sentence of Szyperski [SZY 02] "Components are for composition" illustrates this case perfectly. By definition, a component must have a design specifically established to support the potential composition to allow interoperability. Component models and associated technologies (CORBA *Component Model* CCM [OMG 12], COM + [MIC 13], Fractal [BRU 06], etc.) exist to provide specific development and deployment frameworks needed to support composition patterns. Such models impose component formats in terms of construction of codes and deployment rules [CRN 11]. Thus, the CBSE strengthens the control of composability and clearly formalizes the associated processes. Ultimately, this formalization raises the solid foundation needed for opportunities of automation. Part of the software community has therefore been redirected to the dynamicity concern as the predominant aspect. Thus, SOSE has been developed from the experience gained of objects and components; however, at the outset, it focused on how to improve the dynamicity. The SOSE seeks to provide an appropriate response to highly volatile environments and thus overcome the constraints imposed by the general purpose of the CBSE.

Figure 1.7 summarizes these displacement-related concerns. Research from the OOSE focuses mainly on reuse and discusses some composability and dynamicity. The CBSE focuses on composability, which strengthens reusability and also seeks to automate its processes. The SOSE focuses mainly on the dynamicity of existing processes to ensure reuse and composition. As for the AOSE, it gave more importance to the dynamicity without significant improvement of composability factors and reusability factors; however, it focuses on cooperation and coordination of agents to solve a problem.

The direct comparison following these three quality factors between object-oriented, component-based, agent-oriented and service-oriented paradigms (which is the more reusable, more modular, and more dynamic) is very difficult to establish because it depends on the perspective of the one who compares. The

results vary depending on the contexts in which he is positioned and he positions each of the four paradigms namely; object-oriented, component-based, agent-oriented and service-oriented software entities. For example, from the point of view of a low-level developer, an object will be easier to reuse than a service, whereas conversely, from a business perspective, a very high level service will be more easily reusable.

Thus, our conceptual comparison framework attempts to take this reality into account by providing these users with all the information required to express their own analyzes and qualitative comparison. These qualitative factors are based on a classification of the material provided by the paradigms, which we grouped by their qualitative criteria.

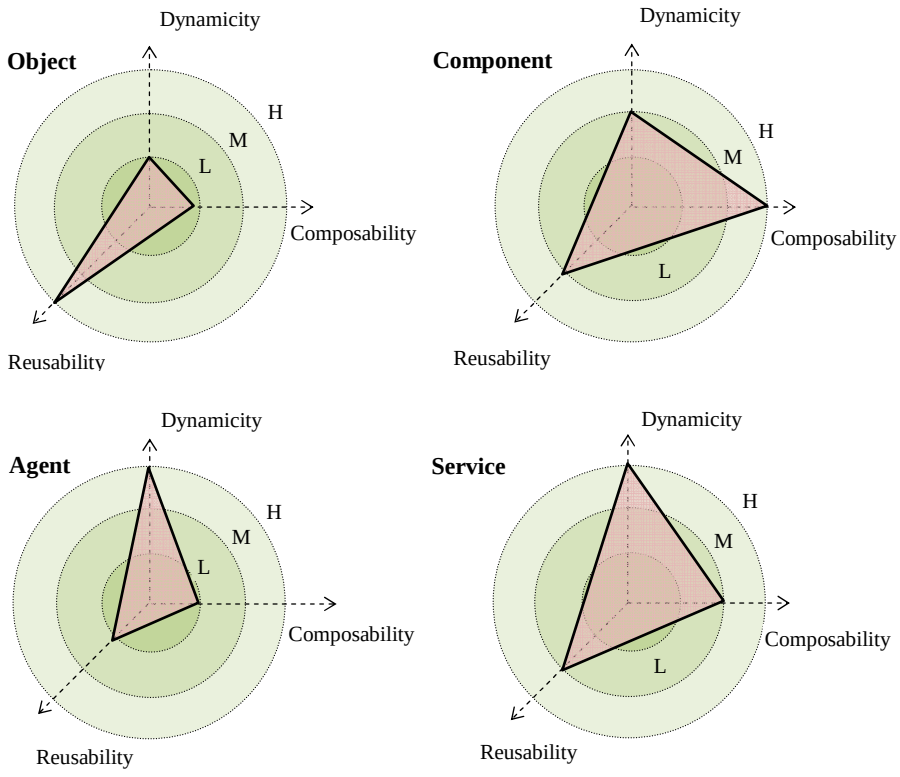

Figure 1.7. *Evolution of overall concerns between paradigms (L: Low, M: Medium, H: High)*

1.4.3.1. *Qualitative criteria for comparing development paradigms*

We have identified eight main qualitative criteria that are common to all software development paradigms. These criteria have a significant impact on the overall quality of the system development process produced as a result:

- *Explicit architecture:* capacity of a paradigm to define clear architectural views of an application, i.e. to provide the means to identify and explain the functions associated with the products that make up the application as well as the processes between these products.
- *Communication abstraction:* capacity of a paradigm to abstract the communication between functions of applications and to learn and understand these communications from one tenant so they can be easily handled.
- *Expressive power:* is the expressive potency of a paradigm in terms of capacity and optionality of creation. It is based on the number of concepts and processes provided to specify, develop, handle, implement and execute applications;
- *Loosely coupling:* is the potential reduction between product-process dependencies.
- *Evolution capacity:* this is the potential of a paradigm to evolve its products and processes. It is based on analysis and judgment value considered on the different processes that support these evolutions and their targets.
- *Owner's responsibility:* this corresponds to the assignment of responsibilities between suppliers and consumers. These responsibilities focus on the reused software entities in terms of development, quality of service, maintenance, deployment, execution and management. This distribution reflects the degree of freedom granted to consumers by the supplier.
- *Concurrency:* in resource-intensive applications that have a demanding need of computational power, concurrency is the most promising solution. Further, concurrency is also highlighted by the recent progress on the hardware side such as the introduction of *multi-core* processors and *graphic cards* with parallel processing capabilities. Mainly, the challenges of concurrency are preserving consistency, prevention against deadlock as well as prevention of race condition dependant behavior.
- *Distribution:* different classes of distributed applications exist according to where the data, the users or computation are distributed. As an example of these classes, we have the *client/server* applications (CS) as well as *peer-to-peer* (P2P) computing applications. The challenges of distribution are

manifold. Among the major concerns of distribution we have future extensions and interoperability, which are often hampered by heterogeneous infrastructure component. In addition, the different scenarios of most applications are nowadays increasingly dynamic with a flexible set of interacting components.

1.4.3.2. *Comparison between paradigms*

Table 1.2 shows the values assigned to the eight criteria to assess the differences between the OOSE, the CBSE, the SOSE and the AOSE. The results are given following three levels of importance (high, medium, low), which are awarded for each criterion and express our analysis of the four paradigms. This comparison establishes a relative assessment between the paradigms (relative to each other).

	Paradigms				
	Quality criteria	**Object**	**Component**	**Service**	**Agent**
1	Explicit architecture	L	M	M	L
2	Communication abstraction	L	M	H	M
3	Expressive power	H	M	L	M
4	Loose coupling	L	M	M	H
5	Evolution capacity	L	H	M	M
6	Owner's responsibility	L	M	H	M
7	Concurrency	L	M	H	H
8	Distribution	L	M	H	H

Table 1.2. *Comparison of development paradigms (L: Low, M: Medium, H: High)*

Software architecture is the cornerstone criterion for the CBSE and the SOSE, unlike the OOSE and the AOSE, which have not taken this concept in their initial definition. To fill this gap, in UML 2.0, OMG introduced the concept of component and connector to describe a software architecture based on the object-oriented mechanisms.

In communication abstractions, the SOSE provides the best communication abstraction based on the encapsulation provided by the services in addition to the isolation of communications in a collaborative scheme. In CBSE, communications are located within different connectors that share the overall behavior. The fine granularity of the object-oriented paradigms worsens this drawback due to the explosive number of collaborations between objects, which is mainly due to the multiple method calls between objects.

Loose coupling is a key issue for the different paradigms. Object-oriented systems involve a set of strongly coupled classes while the CBSE, the SOSE and the AOSE target a reduction of this coupling to make it looser.

Regarding the expressive power, the OOSE handles a large number of concepts such as inheritance, levels of abstraction, levels of description, granularity, reflection, etc. These concepts are expressed through different programming languages such as Java and C++. The CBSE is largely inspired by the object-oriented paradigm, but it has not yet reached the same level of maturity. Finally, the SOSE has the lowest expressive power, because it combines the same shortcomings, plus inaccuracies on levels of abstraction, as the component-based paradigm.

The evolution capacity is directly related to the notion of explicit architecture. Software architecture can be depicted on a graph of nodes and edges. Evolution processes can be grouped according to their target: nodes, edges, or the graph. Typically, the OOSE does not have this notion of explicit architecture. The OOSE evolution process focuses only on nodes and edges. Instead, the CBSE and the SOSE handle the concept of explicit architecture and therefore offer evolution process on three targets. However, the most important maturity of the CBSE and the explicit management of the levels of abstraction have enabled the community to go further and to propose evolution processes at the meta-architecture and meta-meta-architecture levels.

Owner's responsibility: The SOSE pushes the owner's responsibility to the maximum where the service provider is responsible for the development, quality of service, maintenance, deployment, execution and management. On the contrary, the CBSE shares responsibilities at the deployment level where the customer becomes responsible for instantiating the component in its implementation, execution and management. In OOSE, the class is typically in white box implementation where the customer is free to manipulate it at will but they have full responsibility of the class.

Concurrency and distribution: AOSE is built around the aspects of concurrency and distribution. These two criteria have appeared in a number of important research studies and have led to the emergence of distributed artificial intelligence (DAI). With this new approach, the work is done by a group of agents, which act in the same environment and must sometimes resolve conflicts caused by this distribution of expertise.

The analytical method used can only establish a relative order between the paradigms compared, where one paradigm is more effective than the other on a particular criterion. However, in the current framework, the results obtained are

limited to relative hierarchies. We believe that this comparison framework between the four paradigms is a step in their qualitative assessment process.

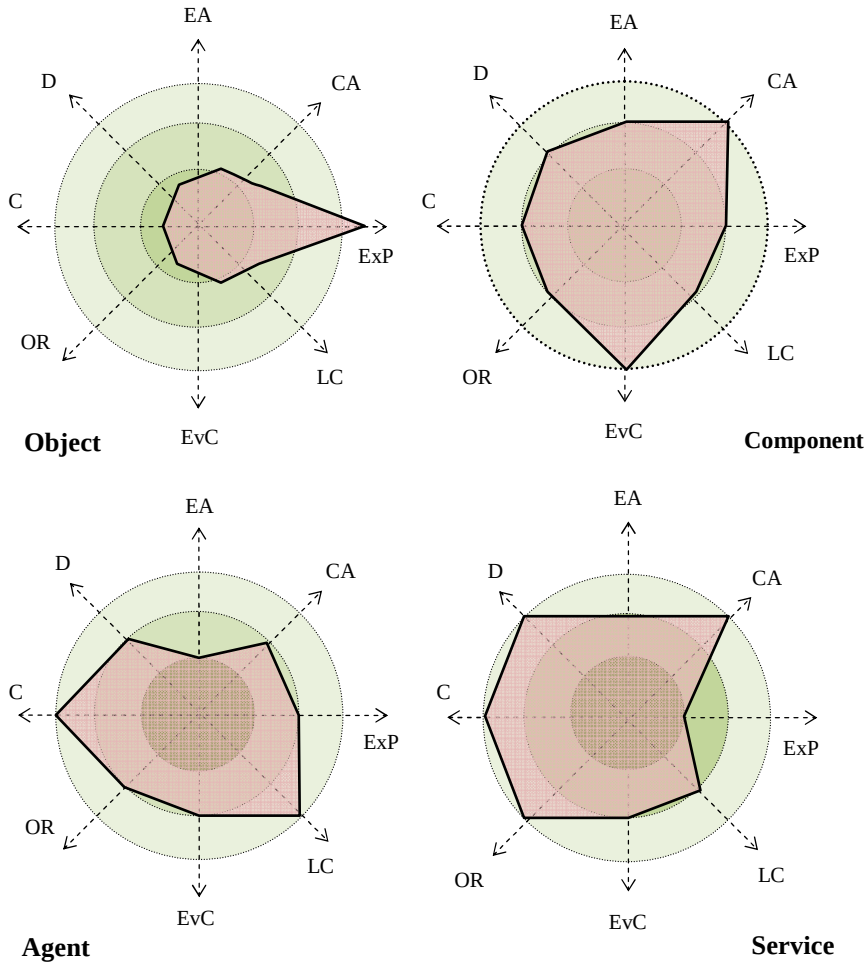

Figure 1.8. *Comparison of criteria with respect to the four paradigms (EA: Explicit Architecture; CA: Communication Abstraction; ExP: Expressive Power; LC: Loose Coupling; EvC: Evolution Capacity; OR: Owner's Responsibility; C: Concurrency; D: Distribution)*

Figure 1.8 shows the use of the eight criteria to assess differences between OOSE, CBSE, SOSE and AOSE. The results are given in three levels of importance (low, medium, high), which are awarded for each criterion and express our analysis of the current status of the three paradigms. Also note that this figure (1.8) represents a graphic interpretation of data displayed in Table 1.2.

1.4.3.3. *Perspective of qualitative analysis*

The conceptual framework that we propose is built to make way for the definition of the user's own assessment of qualitative perspectives. The chosen approach is that the user expresses their knowledge by specifying the perspective through which they want to study the four paradigms being compared. A particular perspective corresponds to the user's focus on a specific factor. It defines a formula for evaluating this factor by combining the results received from the previous comparison, i.e. following the steps of our eight quality criteria.

A qualitative perspective is the combination of:

- The chosen factor to compare the paradigms.
- The expression of the user's expertise in relation to this factor.

Thus, we define a standard formula, which models this ability to customize:

$$\text{Quality} = Q\ (\alpha_1, \alpha_2, \alpha_3, \alpha_4, \alpha_5, \alpha_6, \alpha_7, \alpha_8)$$

The α_i coefficients express the importance of the eight quality criteria, which is given by the user with the target factor. The Q function defines how the coefficients are combined along with the measurements of properties.

A perspective is therefore a qualitative window based on the eight criteria and their results. As an illustration, we assess the four paradigms following our personal viewpoint on the three selected quality factors: reusability, composability and dynamicity that represent the core concerns of OOSE, CBSE, SOSE and AOSE paradigms.

1.4.3.3.1. Example of qualitative perspectives: reusability, composability, dynamicity

In Figure 1.9 we divide the quality criteria based on the impact they have on the different quality factors.

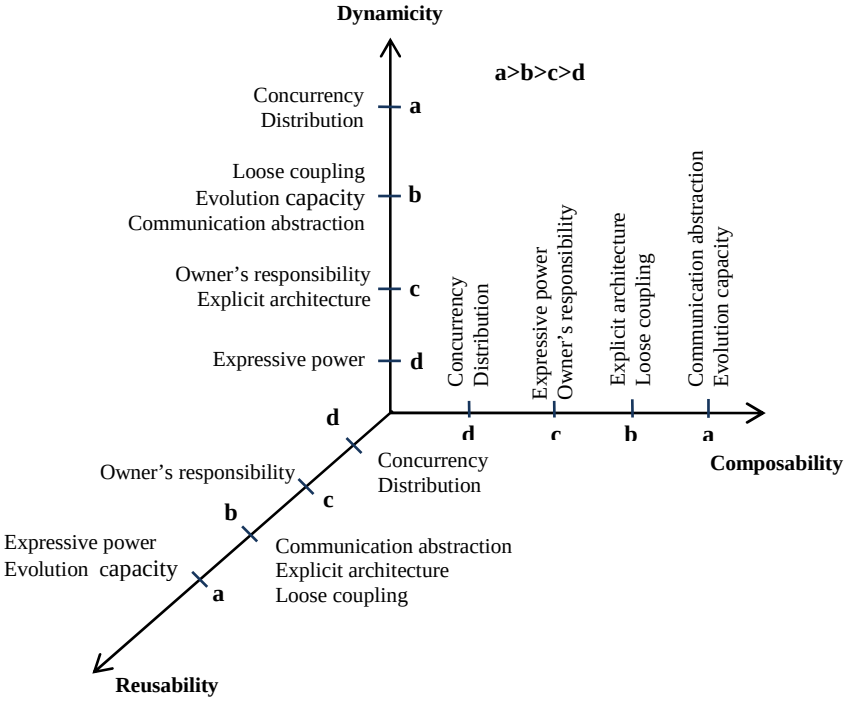

Figure 1.9. *Perspectives expressions of reusability, composability and dynamicity*

Reusability is mainly influenced by the expressive power and evolution capacity with an "a" coefficient, followed by the communication abstraction, the explicit architecture and low coupling with a "b" coefficient, the owner's responsibility with a "c" coefficient and finally the concurrency and the distribution with a "d" coefficient, where (a, b, c, d) represent coefficients of importance of each criterion with respect to a quality factor, with (a> b> c> d). From there, we define a set of formulas that combines this distribution and the results of the previous classification of the four paradigms. To calculate a numerical measure, we associate a weight to each level of Figure 1.8 of 1, 2 and 3 for low, medium and high levels respectively. For example, for the reusability of the factor (r) of each paradigm, we obtain the assessment of the quality function Q_r:

$$Q_{r,\,object} = b\alpha_1 + b\alpha_2 + 3a\alpha_3 + b\alpha_4 + a\alpha_5 + c\alpha_6 + d\alpha_7 + d\alpha_8$$

$$Q_{r,\,component} = 2b\alpha_1 + 3b\alpha_2 + 2a\alpha_3 + 2b\alpha_4 + 3a\alpha_5 + 2c\alpha_6 + 2d\alpha_7 + 2d\alpha_8$$

$$Q_{r,\ agent} = b\alpha_1 + 2b\alpha_2 + 2a\alpha_3 + 3b\alpha_4 + 2a\alpha_5 + 2c\alpha_6 + 3d\alpha_7 + 2d\alpha_8$$

$$Q_{r,\ service} = 2b\alpha_1 + 3b\alpha_2 + a\alpha_3 + 2b\alpha_4 + 2a\alpha_5 + 3c\alpha_6 + 3d\alpha_7 + 3d\alpha_8$$

Composability is generally influenced by the communication abstraction and evolution capacity with an "a" coefficient, then, by the explicit architecture and loose coupling with a "b" coefficient, the owner's responsibility and expressive power with a "c" coefficient and finally the concurrency and distribution with a "d" coefficient where (a> b> c> d). From there, we define a set of formulas that combines this distribution and the results of the previous classification of the four paradigms. To calculate a numerical measure, we associate a weight to each level of Figure 1.8 of 1, 2 and 3 for low, medium and high levels respectively. For example, for the composability factor (cp) of each paradigm we obtain the assessment of the quality function Q_{cp}:

$$Q_{cp,\ object} = b\alpha_1 + a\alpha_2 + 3c\alpha_3 + b\alpha_4 + a\alpha_5 + c\alpha_6 + d\alpha_7 + d\alpha_8$$

$$Q_{cp,\ component} = 2b\alpha_1 + 3a\alpha_2 + 2c\alpha_3 + 2b\alpha_4 + 3a\alpha_5 + 2c\alpha_6 + 2d\alpha_7 + 2d\alpha_8$$

$$Q_{cp,\ agent} = b\alpha_1 + 2a\alpha_2 + 2c\alpha_3 + 3b\alpha_4 + 2a\alpha_5 + 2c\alpha_6 + 3d\alpha_7 + 2d\alpha_8$$

$$Q_{cp,\ service} = 2b\alpha_1 + 3a\alpha_2 + c\alpha_3 + 2b\alpha_4 + 2a\alpha_5 + 3c\alpha_6 + 3d\alpha_7 + 3d\alpha_8$$

Dynamicity is mainly influenced by concurrency and distribution with an "a" coefficient, then the communication abstraction and evolution capacity and loose coupling with a "b" coefficient, the explicit architecture and owner's responsibility with a "c" coefficient, and finally the expressive power with a "d" coefficient where (a> b> c> d). From there, we define a set of formulas that combines this distribution and the results of the previous classification of the four paradigms. To calculate a numerical measure, we associate a point at each level of Figure 1.8 with 1, 2 and 3 for low, medium and high levels respectively. For example, for the dynamicity factor (d) of each paradigm, we obtain the assessment of the quality function Q_d:

$$Q_{d,\ object} = b\alpha_1 + a\alpha_2 + 3c\alpha_3 + a\alpha_4 + a\alpha_5 + b\alpha_6 + a\alpha_7 + a\alpha_8$$

$$Q_{d,\ component} = 2b\alpha_1 + 3a\alpha_2 + 2c\alpha_3 + 2a\alpha_4 + 3a\alpha_5 + 2b\alpha_6 + 2a\alpha_7 + 2a\alpha_8$$

$$Q_{d,\ agent} = b\alpha_1 + 2a\alpha_2 + 2c\alpha_3 + 3a\alpha_4 + 2a\alpha_5 + 2b\alpha_6 + 3a\alpha_7 + 2a\alpha_8$$

$$Q_{d,\ service} = 2b\alpha_1 + 3a\alpha_2 + c\alpha_3 + 2a\alpha_4 + 2a\alpha_5 + 3b\alpha_6 + 3a\alpha_7 + 3a\alpha_8$$

In summary, the conceptual framework provides a comparative picture between object-oriented, component-based, agent-oriented and service-oriented paradigms. These categories identify the important software development paradigm characteristics and provide a common applicable framework to assess the OOSE, the CBSE, the AOSE and the SOSE in a fair manner. This assessment

is quantitative and qualitative in nature and offers an overall understanding of their similarities and differences. However, the quantitative assessment described is only relative, i.e. it establishes a relationship of superiority between the paradigms but does measure neither their values nor their differences. The example of perspectives shown above resulting from this relative assessment provides therefore relative results.

Figure 1.10 summarizes the functioning of our conceptual framework for comparison. The quantitative aspect represents the processes and products of the corresponding paradigms that are the pillars of the eight properties. These properties characterize the quality criteria and serve as vocabulary to users to express their own perspectives on the qualities that concern them.

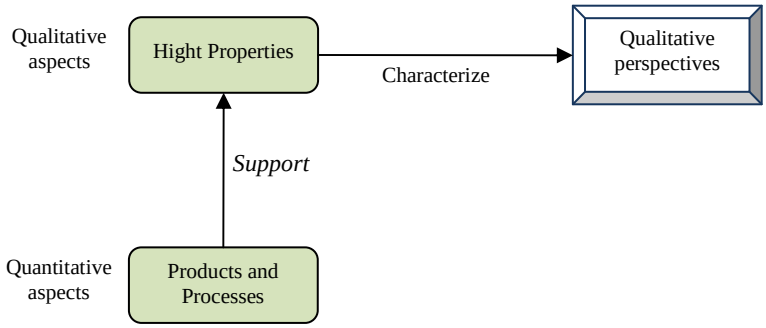

Figure 1.10. *Overall functioning of the conceptual framework*

1.5. Approaches for integrating development paradigms

Several alternative approaches for integrating paradigms were categorized by the entities they aim to combine (object, agent, component, and service). These approaches are based either on conceptual proposals or combinations of technical and conceptual proposals (e.g., *middleware* see Table 1.3).

The active object-oriented model is an example object-oriented and agent-oriented combination which represents an object that conceptually runs on its own thread and provides an asynchronous execution of method invocations. It can thus be considered as a higher-level concept for concurrency of the object oriented systems [SUT 05]. Further, language extensions to support concurrency and distribution were proposed. Eiffel [MEY 93] is an influential proposal in this direction.

Also in the area of agent-oriented and component-based paradigms, combinations proposals can be found. CompAA [ANI 08], SoSAA [DRA 09] and Agent Components [KRU 03] try to extend the agent-oriented paradigm with the concepts and mechanisms of the component-based paradigms. In CompAA, a component-based model is extended with adaptation points for services. These adaptation points allow for service selection at runtime according to the specifications of the functional and non-functional properties in the model. This flexibility is achieved by the addition of an agent for each component, which is responsible for selecting the service at runtime. The SoSAA architecture consists of a base layer with standard component system, and a layer of agents, which controls the base layer to perform reconfigurations as an example of control. In Agent Components, the agents are slightly rendered as components connected together using ports with predefined communication protocols.

Approaches that combine the agent-oriented paradigm with SOA are primarily motivated by the need for dynamic service composition where agents are used to dynamically search and select services during the execution. These approaches deal mainly with aspects linked to the semantic description and research of services, but do not aim at a paradigm integration by itself. As examples, we have the agent-oriented invocation of services using the WSIG[6] component (Web Service Integration Gateway) of the JADE platform, or the code generation approach led by the PIM4Agents model [ZIN 08] and workflow approaches such as WADE or JBees [EHR 05]. Agents are useful in achieving flexible and adaptable workflows using dynamic composition techniques based on negotiation and planning mechanisms.

We also find other approaches that combine the agent, component and object paradigms. ProActive [BAU 09] and AmbientTalk [VAN 07] are two recent approaches in this category, which provide strong conceptual foundations and ready-to-use framework.

An approach in the context of software engineering has emerged under the name of *Service Component Architecture* (SCA) [MAR 09]. It was proposed by several major suppliers of the software industry, including IBM, Oracle and TIBCO. The SCA combines service-oriented architecture (SOA) with the component-based paradigms to provide SCA components that communicate via services.

Braubach and Pokahr [BRA 12] propose the concept of active components during the development of a distributed system project. The active component-

6. http://jade.tilab.com/.

based paradigm is proposed in the framework of an approach to reconcile ideas as well unifying the contributions of the object-oriented, component-based, service-oriented and agent-oriented concepts using a common conceptual perspective. The proposed paradigm is supported on one side by a programming model, which allows the development of systems with active components using XML and Java, and on the other hand by a *middleware* infrastructure, which directs a transparent distribution of the components and provides useful development tools. The active components are an upgrade of the SCA by adding the agent-oriented element in the SCA. The general idea is to transform passive components of the SCA to providers and service consumers, who act independently to better reflect real-world scenarios that are made up of different active stakeholders.

Aboud [ABO 12] proposes a metamodel combination approach called CASOM (Component Agent Service Oriented Model) allowing the specification of applications composed from a set of interoperable agents, components and services in coherent scheme.

	Combined Paradigms			
Approaches and Languages	**Object**	**Component**	**Agent**	**Service**
Programming languages, Java, C#, etc.	x			
Application Server (JBOSS, Glassfish) Component Specification by ADL		x		
Web service and *Business Process Specifications*				x
FIPA Agent specifications, Agent-oriented platforms (JADE, Cougaar, etc.)			x	
Eiffel, active objects	x		x	
WSIG, WADE, PIM4Agents, JBees, etc.			x	x
Fractal, Java EE, OSGI, .Net	x	x		
Service Component Architecture (Passive SCA)		x		x
CompAA, SoSAA, AgentComponents		x	x	
Active Components [BRA 12], CASOM [ABO 12]		x	x	x

Table 1.3. *Approaches for integrating paradigms*

1.6. Summary and discussion

We recall that our purpose through this chapter is to provide a cross-sectional view of the four paradigms namely OOSE, CBSE, SOSE and AOSE.

The CBSE and the SOSE have two different points of view on the relationship between the customer and the supplier. The SOSE comes from certain functional requirements of specific application domains that have specific needs in terms of agility and adaptability, while the CBSE is defined for a larger purpose. The Service-oriented and component-oriented paradigms have a very high granularity. However, a service-oriented paradigm is generally of higher granularity than a component-based paradigm.

An *ad hoc* distinction between the agent-oriented and object-oriented paradigms is that:

- The agent-oriented paradigms are more independent than the object-oriented paradigms.
- The agent-oriented paradigms have a flexible, reactive, proactive and social behavior.
- The agent-oriented paradigms have at least one control *thread* but may have more.

Agent-oriented paradigms can be considered as active objects that encapsulate both their state and behavior, and they can communicate by exchanging messages. The agent-oriented paradigms represent a mechanism of natural abstraction to decompose and organize complex systems just as the object-oriented did before them. An agent-oriented paradigm is a system of rational decision-making: we need an agent to be able to have a reactive and proactive behavior and to be able to perform the interweaving of these two types of behavior, if necessary.

Object-oriented paradigms are generally passive in nature: they need to receive a message before they become active. Although object-oriented paradigms encapsulate their state and behavior, they do not encapsulate behavioral activation. Thus, any object-oriented can call any another object's public method. Once the method is called, the corresponding actions are performed.

While this approach is sufficient for small applications in cooperative and well-controlled environments, it is not suitable for large, concurrent or competitive environments because the entire burden of invocation behavior will be charged to the customer. However, it will be better if the invocation action becomes a process of mutual consent.

According to these relevant observations, under the control of a single organization, software systems must move from one environment towards an open environment in which the system contains organisms that compete with each other.

The object-oriented paradigm fails to provide an adequate set of concepts and mechanisms for modeling complex systems. Individual objects have a fine-grained behavioral granularity and the invocation method is a mechanism too primitive to describe the different types of interactions that may occur.

An approach that is closely linked to that of the object-oriented approach is based on software components. The most important factor behind the component systems is the ultimate goal of software reuse. Essentially, a component-based model enables developers to create and combine software as units of deployment.

According to the description introduced for the component-based approach, we can say that it is a *top-down* approach to the object-oriented approach; therefore, it inherits all the properties of the object-oriented paradigms. The similarities between the object-oriented and agent-oriented paradigms are also present between the agent-oriented and component-based paradigms.

However, the components are not autonomous in the sense of the definition of the independence of agents; in addition, just as the object-oriented paradigms, the component-based paradigms do not have direct notions of responsiveness, pro-activity and social behavior. The service-oriented approach is considered an evolution of the component-based approach enriched by the principles of dynamicity, discovery and composition. To sketch the basic ideas of development we propose a characterization tree map shown in Figure 1.11.

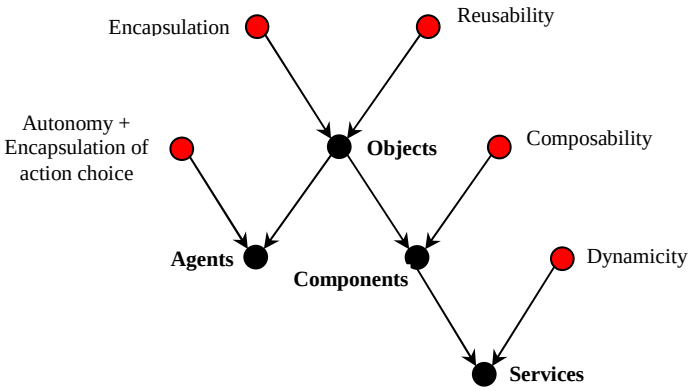

Figure 1.11. *Development of paradigms*

1.7. Conclusion

In this chapter, we have shown a conceptual comparison framework between the four software development paradigms (object-oriented, component-based, agent-oriented and service-oriented). The main objective of this chapter is to provide a clear specification of the conceptual differences between these paradigms and their contribution to the description of software architectures.

Our analysis is carried out through a *top-down* approach, which focuses first on the conceptual aspects of the different paradigms before developing the qualities that result from them. Decisions and choices on this comparison framework have a common goal to make it as generic as possible and independent of any description of software architecture paradigm. Therefore, we endeavor to increase its potential for reusability and ability to be applied to other areas of software engineering.

The proposed framework is based on two dimensions: quantitative and qualitative, where product and process represent the quantitative dimension. This dimension introduces the concepts defined by each paradigm required in the description of a software architecture. Meanwhile, the qualitative dimension defines eight properties, which groups the elements that influence the quality of software architecture.

Finally, the qualitative dimension allows users to express their own quality factors using the eight selected quality criteria. This capacity to customize is

provided by the notion of qualitative perspective that reflects the expertise of the user. In fact, a perspective allows him to communicate their understanding of a target quality through the way that they exploit and combine these properties.

1.8. Bibliography

[ABO 12] ABOUD N. A., Services-oriented integration of component and organizational multi-agents models, Thèse de doctorat, Université de Pau et des pays de l'Adour, 2012.

[AMI 09] AMIRAT A., OUSSALAH M., « First-class connectors to support systematic construction of hierarchical software architecture », *Journal of object technology*, vol. 8, n° 7, p. 107-130, 2009.

[AND 08] ANDRÉ P., ARDOUREG., ATTIOGBÉC., « Composing components with shared services in the KmeliaModel », *Software composition*, p. 125-140, 2008.

[ANI 08] ANIORTÉ P., LACOUTURE J., « CompAA : A self-adaptable component model for open systems », *15th IEEE International conference and workshop on engineering of computer based systems (ECBS'08)*, p. 19-25, 2008.

[BAR 06] BARBIER F., « An enhanced composition model for conversational enterprise JavaBeans », *Proceedings of the 9th international conference on component-based software engineering (CBSE'06)*, p. 344-351, 2006.

[BAU 09] BAUDE F., CAROMEL D., DALMASSO C., DANELUTTO M., GETOV V., HENRIO L., PREZ C., « Gcm: a grid extension to fractal for autonomous distributed components », *Annals of Telecommunications*, vol. 64, p. 5-24, 2009.

[BEI 07] BEISIEGEL M., BOOZ D., EDWARDS M., HERNESS E., KINDER S., « Software components: coarse-grained versus fine-grained », *IBM Developer Works*, 2007.

[BIA 07] BIANCO P., KOTERMANSKI R., MERSON P., Evaluating a services-oriented architecture, Technical report, Software engineering institute, Carnegie Mellon University, 2007.

[BOU 92] BOURON M.T., Structures de communication et d'organisation pour la coopération dans un univers multi-agents, Thèse de doctorat de l'université Paris 6, 1992.

[BRA 12] BRAUBACH L., POKAHR A., « Developing distributed systems with Active Components and Jadex », *Scalable computing: practice and experience*, vol. 13, n°2, 2012.

[BRE 07] BREIVOLD H.P., LARSSON M., « Component-based and services-oriented software engineering: Key concepts and principles », *Proceedings of the 33rd EUROMICRO conference on software engineering and advanced applications*, p. 13-20, 2007.

[BRU 06] BRUNETONE., COUPAYET., LECLERCQM., QUÉMA V., STEFANIJ.B., « The FRACTAL component model and its support in Java », *Software practice and experience*, vol. 36, n° 11-12, p. 1257-1284, 2006.

[CAM 83] CAMMARATA S., MCARTHUR D., STEEB R., « Strategies of cooperation in distributed problem-solving », *International joint conference on artificial intelligence (IJCAI)*, p.767-770, 1983.

[CAS 03] ALONSO G., CASATI F., KUNO H., MACHIRAJU V., *Web services: concepts, architectures and applications*, Springer, Berlin, 2003.

[CAV 09] CAVALLARO L., NITTOE.D., PRADELLAM., « An automatic approach to enable replacement of conversational services », *ICSOC/ServiceWave*, p. 159-174, 2009.

[COX 91] COX B.J., NOVOBILSKI A.J., *Object-oriented programming: An evolutionary approach*, 2nd edition, Addison Wesley, Boston, 1991.

[CRN 06] CRNKOVIC I., CHAUDRON M., LARSSON S., « Component-based development process and component lifecycle », *International conference on software engineering advances*, p. 44, 2006.

[CRN 11] CRNKOVIĆ I., CHAUDRON M., SENTILLES S.,VULGARAKIS A., « A classification framework for software component models », *Journal : IEEE Transactions on software engineering*, vol. 37, n° 5, p. 593- 615, 2011.

[DRA 09] DRAGONE M., LILLIS D., COLLIER R., O'HARE G., « SoSAA: A framework for integrating components & agents », *Symposium on applied computing*, ACM Press, 2009.

[DUR 89] DURFEE E.H., *Coordination of distributed problem solvers*, Kluwer Academic, Boston,1989.

[DUS 05] DUSTDAR S., SCHREINER W., « A survey on web services composition », *International journal of Web and grid services*, vol. 1, n° 1, p. 1-30, 2005.

[EHR 05] EHRLER L., FLEURKE M., PURVIS M., TONY B., SAVARIMUTHU R., « Agent-based workflow management systems », *Journal of information systems and e-business management*, vol. 4, p. 5–23, 2005.

[ERI 08] ERICKSON J., SIAU K., « Web services, services-oriented computing, and services-oriented architecture: separating hype from reality », *Journal of database management (JDM)*, vol. 19, n° 3, p. 42-54, 2008.

[FER 03] FERBER J., GUTKNECHTO., MICHELF., « From agents to organizations: An organizational view of multi-agents systems », *Agent-oriented software engineering* (AOSE), p. 214-230, 2003.

[GAR 97] GARLAN D., MONROE R.T., WILE D., « Acme: an architecture description interchange language », *Proceedings of the 1997 conference of the centre for advanced studies on collaborative research (CASCON'97)*, p. 7, 1997.

[GAS 92] GASSER L., BRIOT J. P., « Object-based concurrent programming and distributed artificial intelligence », in N. Avouris, L. Gasser (dir.), *Distributed artificial intelligence: Theory and praxis*, p. 81-107, Kluwer, Norwell, 1992.

[GEE 08] GEEBELENK., MICHIELSS., JOOSENW., « Dynamic reconfiguration using template based web service composition », *Proceedings of the 3rd workshop on middleware for service oriented computing (MW4SOC'08)*, p. 49-54, 2008.

[GOL 83] GOLDBERG A., ROBSON D., *Smalltalk-80: The language and its implementation*, Addison-Wesley, Boston, 1983.

[HEI 01] HEINEMAN G.T., COUNCILL W.T., *Component-based software engineering: Putting the pieces together*, Addision Wesley professional, Boston, 2001.

[HEW 73] HEWITT C., BISHOP P., STEIGER R., « A universal modular actor formalism for artificial intelligence », *In the 3rd International joint conference on artificial intelligence (IJCAI'73)*, 1973.

[HEW 77] HEWITT C., « Viewing control structures as patterns of passing messages », *Journal of artificial intelligence*, vol. 8, n°3, p. 323-364, 1977.

[HEW 11] HEWITT C., « Actor model of computation: Scalable robust information systems », *Proceedings of inconsistency robustness*, 2011.

[HOC 11] HOCK-KOON A., Contribution à la compréhension et à la modélisation de la composition et du couplage faible de services dans les architectures orientées services, Thèse de doctorat, Université de Nantes, 2011.

[HYA 96] HYACINTH S. N., « Software agents: An overview », *Knowledge engineering review*, vol. 11, n°3, p. 205-244, 1996.

[JAC 05] JACOB D., « Enterprise software as service », *Queue - enterprise distributed computing*, vol. 3, n° 6, p. 36-42, 2005.

[JEN 01] JENNINGS N.R., « An agent-based approach for building complex software systems », *Communications of the ACM*, vol. 44, n° 4, p. 35-41, 2001.

[KAY 93] KAY A. C., « The early history of Smalltalk », *ACM SIGPLAN Notices*, vol. 28, n° 3, p. 69-95, 1993.

[KIT 96] KITCHENHAM B., PFLEEGER S.L., «Software quality: the elusive target», *IEEE Software*, special issues section, n° 1, p. 12-21, 1996.

[KRU 03] KRUTISCH R., MEIER P., WIRSING M., « The agent component approach, combining agents, and components », *1st German conference on multi-agent system technologies (MATES)*, p. 1-12, Springer, Berlin, 2003.

[MAR 09] MARINO J., ROWLEY M., *Understanding SCA (Service component architecture)*, 1st edition, Addison Wesley professional, Boston, 2009.

[MCI 68] MCILROY D., « Mass-produced software components », in J.M. Buxton, P. Naur, B. Randell (dir.), *Software engineering concepts and techniques*, p. 88-98, NATO Science Committee, 1968.

[MEY 93] MEYERB., « Systematic concurrent object-oriented programming », *Communication ACM*, 36, p. 56-80, 1993.

[MIC 13] MICROSOFT COM (Component Object Model) Technology, www.microsoft.com/com/default.mspx, 2013.

[MOH 08] MOHAMED A., ZULKERNINE M., « At what level of granularity should we be componentizing for software reliability? », *11th IEEE High assurance systems engineering symposium (HASE'08)*, p. 273-282, 2008.

[NIT 08] NITTOE.D., GHEZZIC., METZGERA., PAPAZOGLOUM., KLAUS P., « A journey to highly dynamic, self-adaptive service-based applications », *Automated software engineering*, vol. 15, n° 3-4, p. 313-341, 2008.

[OAS 08] OAS, *Reference architecture for service oriented architecture, version 1.0*, 2008, http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html.

[OAS 09] OAS, *Service component architecture assembly model specification, version 1.1*, 2009, http://docs.oasis-opensa.org.

[OMG 12] OMG, Common object request broker architecture (CORBA), formal/2012-11-16, www.omg.org/spec/CORBA/.

[OUS 99] OUSSALAH M. *et al*., *Génie objet*, Lavoisier, Paris, 1999.

[OUS 05] OUSSALAH M. *et al*., *Ingénierie des composants : Concepts, techniques et outils*, Vuibert, Paris, 2005.

[PAP 07] PAPAZOGLOU M.P., HEUVEL W.J., « Service-oriented architectures: Approaches, technologies and research issues », *The VLDB Journal*, vol. 16, p. 389-415, 2007.

[PES 00] PESCHANSKI F., MEURISSE T., BRIOT J.P., « Les composants logiciels : Evolution technologique ou nouveau paradigme? », *Actes de la conférence objets, composants, modèles (OCM'00)*, Nantes, France, p. 53-65, 2000.

[SMI 81] SMITH R.G., DAVIS R., « Frameworks for cooperation in distributed problem-solving », *IEEE Transactions on systems, man and cybernetics*, vol. 11, n° 1, p. 61-70, 1981.

[SOM 04] SOMMERVILLE I., *Software engineering*, 7e edition, Addison Wesley, Harlow, 2004.

[SUT 05] SUTTER H., LARUS J., « Software and the concurrency revolution », *ACM Queue*, vol. 3, n° 7, p. 54-62, 2005.

[STO 05] STOJANOVIC Z., DAHANAYAKEA., *Services-oriented Software System Engineering: Challenges and Practices*, IGI Publishing, Hershey, PA, 2005.

[SZY 02] SZYPERSKI C., *Component software: Beyend object-oriented programming*, Addison-Wesley Professional, Harlow, 2002.

[TAY 09] TAYLOR R.N., MEDVIDOVIC N., DASHOFY E., *Software architecture: Foundations, theory, and practice*, Wiley-Blackwell, Chichester, 2009.

[THE 08] TheSeCSETeam, *Service centric system engineering*, EU Integrated Project, 2008, www.secse-project.eu/.

[VAN 07]VAN CUTSEM T., MOSTINCKX S., BOIX E.G., DEDECKER J., DE MEUTER W., « AmbientTalk: Object-oriented event-driven programming in mobile ad hoc networks », *Chilean computer science society*, p. 3-12, 2007.

[VIN 97] VINOSKI S., « CORBA: Integrating diverse applications within distributed heterogeneous environments », *IEEE Communications magazine*, vol. 14, 1997.

[WEI 91] WEISER M., *The computer for the 21st century*, p. 94-104, Scientific american, New York, 1991.

[WOO 09] WOOLDRIDGE M., *An introduction to multi-agent systems*, 2de edition, John Wiley & Sons, New York, 2009.

[ZEN 03] ZENGL., BENATALLAHB., DUMASM., KALAGNANAMJ., SHENGQ.Z., « Quality driven web services composition », *Proceedings of the 12th international conference on World Wide Web* (*WWW'03)*, p. 411-421, 2003.

[ZIN 08] ZINNIKUS I., HAHN C., FISCHER K., « A model-driven, agent-based approach for the integration of services into a collaborative business process », *Proceeding of AAMAS, IFAAMAS'08*, p. 241-248, 2008.