From Component-based Architectures to Microservices: A 25-years-long Journey in Designing and Realizing Service-based Systems

Giuseppe De Giacomo[0000−0001−9680−7658],
Maurizio Lenzerini[0000−0003−2875−6187],
Francesco Leotta[0000−0001−9216−8502], and
Massimo Mecella[0000−0002−9730−8882]

Sapienza Università di Roma
Dipartimento di Ingegneria Informatica Automatica e Gestionale
{degiacomo,lenzerini,leotta,mecella}@diag.uniroma1.it

Abstract. Distributed information systems and applications are generally described in terms of components and interfaces among them. How these component-based architectures have been designed and implemented evolved over the years, giving rise to the so-called paradigm of Service-Oriented Computing (SOC). In this chapter, we will follow a 25-years-long journey on how design methodologies and supporting technologies influenced one each other, and we discuss how already back in the late 90s the ancestors of the SOC paradigm were there, already paving the way for the technological evolution recently leading to microservice architectures and serverless computing.

Keywords: components · SOC · middleware technologies · microservices · design methodologies

1 Introduction

Divide et impera[1] describes an approach, relevant to many fields, where, to solve a problem, it is required or advantageous to break or divide what opposes the solution. As an example, in computer science and engineering, it is applied at the level of standalone programs, by organizing codes in functions, classes and libraries, but it is also at the basis of how complex business processes, spanning several departments/offices of the same organization and/or different organizations, are implemented by making different parties collaborating on a computer network [1]. Each part is referred to as a *component* and communicates with other components by means of software interfaces.

The first examples of such interfaces were implemented back in 1984, even though request-response protocols were available since late 1960s, using Remote Procedure Calls (RPCs) [2]. Since then, technologies evolved in order to support

[1] This was originally an Anciet Roman socio-political technique; the motto is attributed to Philippus II of Macedon.

more and more distributed, performing and resilient applications, reaching the current peak of modern highly distributed microservice architectures based on cloud infrastructures.

Concurrently, with the increasing complexity of systems, a joint effort from the research community, practitioners and consultants, and industry led to the development of design methodologies and best practices covering the different phases of software development at any granularity, from classes to distributed architectures. The latter are usually described in terms of software components and interfaces and for this reason they were referred to as *component-based architectures*. Design methodologies for component-based architectures date back to the late 90s, but the features of components they currently define is much more powerful than what technologies allowed at that time.

Technologies then evolved into Web-ready (HTTP-based) components, referred to as *Web services*, leading to the emergence of Service-Oriented Computing [3–5].

In this chapter, we discuss how the potentials of that notion of component can be unleashed only nowadays, when effective technologies for distribution and replication are available, and how the modern concept of microservice was already anticipated by those design methodologies.

2 Design of component-based architectures

The concept behind a component-based architecture is simple: a big application is split into autonomous entities, i.e., components potentially realizing on a distributed scale what classes are in object-oriented programming. As such, components share with classes features, e.g., encapsulation, identity and unification of data and functionalities, standard interfaces, and final goals, i.e., reuse and transparency. However, the different scale at which these principles are applied requires a specific design methodology.

Design of component-based architectures has its roots in the design of object-oriented software. After all, and for the sake of simplicity, components can be considered as classes distributed on a network. A basis for object-oriented design already looking ahead to components is provided by the Catalysis approach [6], where a design methodology is proposed based on the Unified Modeling Language – UML. UML diagrams are used to represent precise specifications of use cases where each operation is described in terms of pre- and post-conditions expressed in Object Constraint Language - OCL.

Later, the authors of [7] extend Catalysis with the emerging concept of component technologies, such as Enterprise Java Beans. The methodology is intended to identify components and their interfaces by a rigorous strategy starting from a business process model describing the domain of interest. The analysis of the business process model produces *(i)* a business concept model and an actor model, both described in terms of UML class diagrams with stereotypes, *(ii)* a use case diagram, with a single use-case for each triggering event in the process model, where triggering events are obtained by looking at which concepts and

associations of the business concept model can change, *(iii)* a set of use case descriptions, and *(iv)* a set of quality-of-service requirements for the system.

The next step is to identify interfaces, which are divided into two types: system interfaces and business interfaces. A system interface is associated in particular to each use case identified in the previous step, and this interface contains an operation for each of the tasks of the use case, which is responsibility of the computing system. Business interfaces are obtained from the business concept model identifying core types, i.e., those classes that have an independent existence, not having mandatory associations except with categorizing types. For each of these core types, a business interface is defined, whose methods allow to manipulate instances of the core types and associations to other types in the business concept model. Other interfaces are then provided as input to the design methodology, such as those of pre-existing software systems.

Notably, business interfaces are very similar to *concepts* in conceptual modeling approaches which, in the same period, were investigated in the information systems field. Again, in the same period, the knowledge representation community was starting to investigate formal approaches and possibly automated reasoning for conceptual modeling, through the use of Description Logics - DLs [8]. Further, a bit later, Domain Driven Design [9] – DDD, was proposed as a way to structure systems, and notably *entities* are not dissimilar from concepts and business objects. DDD is considered the starting point of the microservice approach (which we will consider later in the chapter).

Back to [7], after the definitions of interfaces, a single component specification is created for all of the system interfaces and several business components will be defined for each of the core types, each containing the single interface defined for that specific core type. System components will use business components to perform complex tasks, whereas business components perform very basic tasks, thus being very close to the concept of microservice. As business components are derived by UML classes, they also implicitly define the corresponding *information model*, which is an important part of a microservice architecture.

At this point, an initial component-based architecture is available, showing the basic interaction between components. This architecture is then enriched by specifying each operation with pre- and post-conditions.

3 Evolving technologies

In this section, we briefly outline an historical evolution of technologies used to implement component-based architectures and then services. Technologies and design methodologies evolved somewhat independently, with the former often developed by practitioners [10], playing chase one each other. In particular, as it will be discussed in Section 5, the concept of component emerged more and more clearly in recent technologies than in older ones.

3.1 RPC/RMI

Initially, technologies emerged that allowed to abstract the same kind of interaction developers where used to employ in single application programming languages between software modules and classes. They hide to the developer all the challenges related to network communication, providing an experience similar to simply calling a programming language library.

The first example of such technologies is represented by Remote Procedure Calls – RPCs. If request–response protocols date back to the late 1960s, at the very beginning of networking, theoretical proposals of RPCs date to the 1970s, but first commercial implementation only appearing in 1984 [2]. With RPC, in order for a client program to call a servant program, the developer must import a stub library, which takes charge of communication related issues such as marshalling and umarshalling of data. The stub can be automatically generated from a textual description of the interface expressed through an Interface Definition Language – IDL. A seminal implementation of RPCs is represented by Sun's RPC (also known as Open Network Computing – ONC, and based on the C programming language). RPCs have also been adapted to object oriented languages, referred to as Remote Method Invocation – RMI. Java RMI, for example, has been introduced in 1997 with Version 1.1 of Java.

In component-based architectures, which started to require different companies to integrate their functionalities, RPCs found obstacles, as binary protocols involved in the communications were not Web-friendly (i.e., created issues in being filtered by Web firewalls). This was not a big issue when integration was between branches of the same organization, and specific firewall rules could be applied. Nevertheless, as soon as integration started involving different organizations/enterprises, this became a limiting factor.

Currently, gRPC [2], initially developed at Google in 2015, is an open source RPC system that uses HTTP/2 for transport, Protocol Buffers [3] as the interface description language, and generates cross-platform client and servant bindings for many languages. The most common usage scenarios include connecting services in microservice architectures and connecting mobile devices and browser clients to back-end services.

3.2 DCE, object brokers and application servers

RPCs are technologies that can be used to connect components. The support to component architectures anyway is not limited to the simple communication task. Deployment and usage of components may require additional functionalities that are historically provided by software modules referred to as middleware. An initial example of this category of software, which marks the passage from two-tier to three-tier systems, is represented by Distributed Communication Environment [11] – DCE, which is dated back late 1992/early 1993. The DCE

[2] J. Kolhe, S. Kuchibhotlag: RPC Intro, KubeCon + CloudNativeCon 2018, Seattle, USA, December 11 - 13, 2018, https://www.youtube.com/watch?v=OZ_Qmklc4zE

[3] Cf. https://developers.google.com/protocol-buffers

runtime environment supported infrastructural functionalities such as directory, time, thread management and distributed file system, which can be still recognized in modern middleware.

DCE is grounded in RPCs. While object oriented programming aroused, the concept of object seemed more fitting than calls to procedures: object brokers appeared on the landscape, being the first and most famous specification the Common Object Request Broker Architecture – CORBA [12]. Differently from DCE, which also enforces a standard implementation, CORBA only consisted of a specification that was then implemented by different vendors (e.g., Iona's Orbix), thus facilitating its adoption. First CORBA specification dates back to 1991, but widespread adoption started only in 1996, further contributing to the low success of DCE. The most significant competitor in the market of object brokers was represented by Microsoft Distributed Component Object Model – DCOM [13], and its descendant COM+ [14], both extensions to the distributed scenario of the single machine component model adopted on Microsoft Windows.

Both CORBA ad DCOM provide additional functionalities to RMI-based interactions, which resemble and extend those provided by DCE, with a stronger support, for example, to transaction management.

The above technologies easily support a single organization, but show limitations in a multi-organization context. This is the reason why, starting from the early 2000s, the employment of object brokers faded in favour of products having Web technologies and HTTP as main communication and access channel, thus allowing an easier inter-organization integration. These are usually referred to as *application servers* [15]. Most used application servers are based on Java 2 Enterprise Edition – J2EE, or on Microsoft .NET. In J2EE application servers, components are realized as Enterprise Java Beans – EJBs.

In an application server, a component is a software module running inside a container, which provides system-level features, e.g. security, transactions, etc. But in order to have this, components should adhere to a contract with the container: it provides features to the components, which in turn must be developed according to a specific structure (i.e., realizes specific interfaces expected by the container). A developer is therefore in charge of two aspects: to realize the methods required by the *component model* and to realize the specific logic of the application.

CGI/server pages and Javascript. Integral part of an application server is the support for the presentation layer, and in particular for dynamically generated Web pages. Common Gateway Interface - CGI, introduced back in 1993, represented the first way for a Web server to return dynamic content. The term gateway is employed to denote an application, serving the calling one, which allows to access the data of another back-end system (e.g., a database/DBMS). Upon the reception of an external call, the Web server executes an instance of the gateway, which is in charge of gathering information from the back-end system (e.g., through a database query), packing it into an HTML page, and returning it back to the Web server, which in turn sends the response back to the client.

Performance issues intrinsic in CGIs, and related to spawning of a new process for each request, have been solved by including gateways as part of the Web server with technologies such as servlets (introduced in 1997), or with the employment of server page technologies, which embed programming language code inside the HTML pages, such as Active Server Pages – ASP, Java Server Pages – JSP, PHP pages, and ASP.NET pages.

CGIs and related subsequent technologies are mainly intended to provide the presentation layers for final users, to be presented directly in a Web browser. In this case, clients are said to be "thin", because they are only responsible for user interaction and visualization of results. Thought still popular, these kinds of technologies have become less and less frequently employed, while client side technologies such as JavaScript has become more and more employed for the presentation layer. In this scenario, remote systems provide functionalities (as discussed in Section 3.5 as REST services), and the Javascript code embedded in the HTML page on the client directly executes the entire presentation logic. Clients are, in this case, said to be "fat".

3.3 Asynchronous integration

Asynchronous integration between components has been always possible since RPCs, which defined both synchronous and asynchronous interactions between clients and servants. In case of long tasks or busy servants, it is better for the client to continue its job waiting for a completion notification from the servant. The servant enqueues requests from the clients and processes them whenever resources are available. A similar need arises when the client needs no answer to a specific request.

Asynchronous interaction is usually supported by Message Oriented Middleware – MOM. MOMs, which are sometimes integrated as part of more complete solution (e.g., CORBA specified its own messaging system), usually provide two kinds of interaction modes: queueing and publish/subscribe. In publish/subscribe systems, the MOM exposes a set of topics where clients can publish messages, that are then broadcasted to topic subscribers following specific authorization rules. This way it is possible to implement different communication schemas such as one-to-many or many-to-many.

MOMs started to widespread in the mid 90s. Historically, among the most widespread technologies, Java Messaging Service – JMS, should be mentioned. Nowadays they represent a very relevant technology, especially in the field of Big Data ingestion including Internet-of-Things (IoT) applications (cf. AMQP, MQTT, RabbitMQ, Kafka).

3.4 Web (SOAP) services

As already discussed, modern middleware are intended to support integration not only in an intra-organization scenario, but also in an inter-organization one. Communications among different organizations through the Internet can be limited by the presence on the network path of firewalls filtering certain protocols.

As a consequence, in the early 2000s, the idea was to replace protocols typical of objects brokers, directly using TCP as a transport protocol, with textual protocols using HTTP as a transport protocol. The very pragmatic advantage was that most firewalls allow HTTP traffic to pass through. The basic idea was to bind the different functionalities of a component to a specific HTTP URL. Each component, referred to as *Web service*, contains a set of methods that can be called using standard HTTP methods (e.g., GET, POST). The first example of this protocol was Simple Object Access Protocol – SOAP, dating back to 2000, even though the first version recommended by W3C is the 1.2, published in 2003. This started the era of Web services and Service-Oriented Computing [3, 5, 4].

In SOAP, the HTTP request/response body issued to/returned from a specific URL contains an XML envelope consisting of an header and a body. The header contains information about the issued methods and other aspects related to security and encoding. The body contains the actual information exchanged between the two parties. Description and specifications of SOAP services is performed through Web Service Definition Language - WSDL (an IDL for Web services). The specifications that were subsequently built on top of SOAP were wide, covering several different aspects needed for application integration.

Using SOAP for inter-organization integration does not prevent intra-organization integration to be performed at the object broker level. For example J2EE application server still allow to use EJBs for intra-organization, allowing to wrap them as SOAP services for inter-organization integration. Nonetheless, practice became more and more to employ SOAP also for intra-organization integration.

3.5 REST services

SOAP services model the access to a component as a set of high-level functionalities, that can be even very complex. Concurrently (early 2000) to this vision, a different one more focused on resources was developed in [16] under the name of REpresentational State Transfer – REST. The REST approach to services is based on the following considerations:

- operations performed on components can be decomposed in simple CRUD (Create, Read, Update, Delete) operations on resources (e.g., chapters in a book of a book catalogue). The advantage is that it is very immediate to map a resource and CRUD operations to HTTP requests, by making URLs mimicking the hierarchical relationships between resources (e.g., chapters contained in the book with ID 1 can be mapped to the URL `books/1/chapters`) and HTTP actions to CRUD operations;
- basic operations on a resource can be considered inherently stateless, with no need to maintain sessions, which can be implemented at HTTP level.

The previous considerations make the development of REST services much simpler than SOAP ones. Despite the concurrent conception of the SOAP and REST approach, the supports from vendors and the employment of services mostly for integration, made during the 2000s SOAP services more popular than

REST ones. In the current decade, instead, the explosion of the service economy, the widespread adoption of mobile apps and the seamless integration of REST services with HTTP, which made it perfectly suitable for calling a service from a Web browser, led developers to prefer REST services where the HTTP body contains information expressed in JavaScript Object Notation – JSON, which is easily manipulated by JavaScript engines in Web browsers (see Section 3.2 when discussing client-side technologies).

Nowadays, choosing between using SOAP or REST services is still an active source of debate [17], but the growth of cloud and serverless computing is pushing more and more REST services, with SOAP services residually employed for niche business-to-business integration.

3.6 Virtualization, serverless and microservices

In the last years, different technology and business trends drove the evolution of development technologies for component-based architectures.

An increasing trend in companies is to deploy developed services and components on servers not directly owned by them, preferring to deploy on remote machines managed by cloud providers (e.g., Amazon, Google, Microsoft), which offer advanced guarantees such as disaster recovery and maintenance. These servers are most of the time virtualized on physical machines owned by cloud providers. Virtualization allows to easily backup and replicate copies of machines for redundancy and load balancing. The extreme evolution of *virtualization* is *containerization*, which allows to very quickly instantiate new containers, each one generally hosting a single component, to accomodate scalability requirements. *Serverless computing* [18] hides server usage from developers and runs code on-demand automatically scaled and billed only for the time the code is running; if the code corresponds to a pure functional component (Function-as-a-Service: the unit of computation is a function that is executed in response to triggers such as events or HTTP requests), the abstraction seen by the developer perfectly correspond to the deployment.

The current evolution of the digital economy is such that the load of services is no longer predictable. In fact, the number of users that need to call services continuously change, making serverless computing a need for organizations, helping them to efficiently use allocated resources on cloud providers, in turn reducing the costs.

The replication of services is a delicate task, as these services may share and distribute also needed resources, in particular data. Keeping a distributed database is a complex task, made even more complex by distributed transactions. This aspect led, on the one hand, to the increasing employment of NoSQL databases that simplify horizontal distribution (sharding) and replication, and, on the other hand, to the development of very fine-grained services performing almost atomic operations on (very few) resources. These services are referred to as *microservices* and perfectly fit with the concepts behind REST services (even though in principle a microservice can be implemented with a different set of technologies). If more complex operations must be performed,

involving transactions and sessions for example, they are implemented by other (micro-)services at an higher level (coordinating or aggregating them). The development of microservices is nowadays supported by means of specific frameworks such as Spring (Java), .NET core and NodeJS. Microservices developed with these technologies can be easily deployed and managed using DevOps [19] and exploiting cloud/serverless computing.

4 Design of Service-Oriented Architectures

As seen in the previous sections, Service-Oriented Architectures emerged as a style of software design where functionalities are provided to client components by servant application components, through a communication protocol over a network. The service is therefore a discrete unit of functionalities that can be accessed remotely and acted upon and updated independently. The service has some properties [3, 5]:

1. it logically represents a business activity with a specified outcome;
2. it is self-contained;
3. it is a black box for its consumers, meaning the consumer does not have to be aware of the service's inner workings; and
4. it may consist of other underlying services.

Notably, *(i)* property 1 implies the development of "coarse-grained" services as opposed to microservices, *(ii)* properties 2 and 3 are shared with component-based design, and *(iii)* property 4 implies that different services can be used in conjunction to provide the functionality of a large software application, thus giving rise to the attention to composition and orchestration of Web services.

When the focus of the research community and practitioners switched from component-based architectures to Service-Oriented Architectures, whereas the design methods proposed in [7] still remain valid, the interest starts converging on how to specify at design-time the single service interface (system interface in particular as defined in Section 2) and the interactions among them.

For what concerns the first aspect, the proposed approaches focus on the specification of a service as a UML state-transition model, thus highlighting the *conversational/stateful* nature of SOAP Web services [20, 21]. An attempt to systematize all of these approaches, starting from the identification of components, to specification of interfaces has been proposed for SOAs in 2008 by Catalysis Conversation Analysis [22].

In addition to UML diagrams, also BPMN diagrams are employed as input for the design process, as they document the process to be partially or fully automated through services/components. The introduction of BPMN, and the attention to business processes, is a distinctive feature of SOA as it allows to better model interactions, which is the second aspect mentioned above, between parties and, as a consequence, services. These services can belong to the very same organization or to a different organization. When a formal specification of

single services is available, service composition [23], aiming at formal checking, or at automatically composing services aiming at a specific formal goal, is possible.

In automatic composition of services [24], a formal specification of the goal service to be composed is given as an input to a composer, which creates the new service starting from available ones. In [25–27], for example, a goal service is described as a guarded automaton to be synthesized combining the automatons of the single services. Here the idea was to have a repository of Web services, each with a formal specification, that could be composed in order to provide a more complex service. Automatic service composition has a theoretical background in model checking and a technological justification in the presence of discovery services in the SOAP specification and in the emerging trends of semantic Web services [28, 29].

Services obtained through automatic service composition are added themselves to the service repository, making them available to be composed with other services. Unfortunately, so far the availability of such repositories of formal and semantically searchable definitions of Web services has remained quite low, thus limiting the chances to apply service composition.

5 Discussing technologies and methods

Technologies and design methodologies has evolved together over years, with the former influencing the latter and viceversa. Initial technologies available for interfaces between components, namely RPCs, reflect a client/server design approach, where servers were monolithic entities encapsulating presentation, application (a.k.a. business) and resource layers [15]. With the increasing complexity of servers, functionalities of the different layers have been distributed on several different machines in two or three physical-tier architectures. Distribution was initially mainly intended as intra-organization leading to the development of object brokers. Design methodologies followed the very same path with the introduction of component identification and specification methods instead of the classical client/server interaction. An architecture had to be compliant with respect to a specific component model (e.g., EJB, COM+, .NET), and ultimately was designed with a goal in mind: the application logic reuse, i.e., the possibility to employ an already available component in a new context. Initially, HTTP-based technologies such as SOAP and, later REST, wrapped component interfaces over HTTP, reflecting the underlying component models.

If the above considerations seem to be in favour of a balanced progression of technologies and design methods, the definition of component introduced in the early 2000s is much more advanced that the technologies available at that time.

The design method proposed in [7] introduces the concept of system and business components, with the latter basically acting as wrappers for the operations allowed on the underneath information model. In this sense, functionalities exposed by business components already support the separation between business layer and resource layer, which is typical of modern architectures. Additionally, these functionalities are very fine-grained thus resembling modern microservices.

As in those methods the concept of multiple component instances was already in place, it can be argued that design methodologies were, in this case, far beyond technologies available right then.

Modern microservice-based architectures are of course not only the result of design methods, but also of a different and more dynamic software life cycle based on continuous integration/deployment/delivery and DevOps.

Additionally, microservices are nowadays employed not only as a mean to perform CRUD operations on the underneath information (model). Instead, they are also an active part of big data processing pipelines in kappa, lambda and delta architectures [30].

A difference between component-based architectures and modern microservice-based ones is anyway that microservice frameworks are more intended for the infrastracture/communication-level reuse than for reuse of the application logic. Component models had an heavy footprint to support application reuse, in terms of both development effort and deployment infrastructure. Modern microservices are instead very lightweight (e.g., based on Plain Old Java Objects – POJO) and no longer based on component models . This is due to the modern awareness that due to fast changes in technologies and requirements, reuse of application logic is only possible at a very small scale (e.g, software libraries), whereas it is very important to reuse the infrastructure and the communication functionalities.

6 Concluding remarks

If, on the one hand, as discussed in the previous section, microservices represent business interfaces described in the early 2000s by design methodologies, on the other hand, their development in the last years has been fast and wild, leaving in some cases part of the design process uncovered.

As an example, the resource layer of microservices is often represented by NoSQL databases, such as document based ones, which are very suitable for replication and sharding, but whose design methodologies are still somewhat weak with respect to those available with relational databases.

Additionally, microservices are usually implemented as REST services. REST is much lighter than SOA, and employed technologies are not anymore based on component models. If WSDL is a rich formalism, which has been, for example, a trigger for an extensive research on semantic Web services and composition techniques, Open API, which is the most widely employed definition approach for REST services, lacks of support for many advanced features such as searchability, support for composition, etc., thus making the development of similar techniques harder. And whereas those approaches failed in the past, probably due to the too large ambition (the entire Web as domain of interest), nowadays services are employed in basically all sectors and maybe limiting to domain-specific scenarios can demonstrate their potential. As a result, component-based architectures and SOC must nowadays be re-thinked, factoring aspects that are still valid and framing them in a world of lightweight frameworks, fast devel-

opment/integration/deployment, sudden changes in requirements and needs for self-adaptability. For what concerns SOC, in particular, authors in [31] identified, in addition to design and service composition, other two emerging research challenges, i.e., crowdsourcing-based reputation, and the Internet of Things (IoT).

A promising approach may reside in automatic synthesis of programs based on specification. This is now feasible thanks to the employment of artificial intelligence applied to UML and BPMN specifications [32, 33]. This approach will be investigated, for example, in the ERC project WhiteMech (n. 834228), which targets the three scenarios of smart manufacturing, smart spaces and business processes.

References

1. M. Dumas, M. La Rosa, J. Mendling, and H. A. Reijers, *Fundamentals of Business Process Management, Second Edition*. Springer, 2018.
2. A. D. Birrell and B. J. Nelson, "Implementing remote procedure calls," *ACM Transactions on Computer Systems (TOCS)*, vol. 2, no. 1, pp. 39–59, 1984.
3. M. P. Papazoglou and D. Georgakopoulos, "Service-Oriented Computing," *Communications of the ACM*, vol. 46, no. 10, pp. 25–28, 2003.
4. M. P. Papazoglou and W.-J. Van Den Heuvel, "Service-Oriented Architectures: approaches, technologies and research issues," *VLDB journal*, vol. 16, no. 3, pp. 389–415, 2007.
5. M. Papazoglou, *Web services: principles and technology*. Pearson Education, 2008.
6. D. F. D'souza and A. C. Wills, *Objects, components, and frameworks with UML: the catalysis approach*. Addison-Wesley, 1998.
7. J. Cheesman and J. Daniels, *UML components*. Addison-Wesley, 2001.
8. D. Calvanese, M. Lenzerini, and D. Nardi, "Description logics for conceptual data modeling," in *Logics for databases and inf. systems*, pp. 229–263, Springer, 1998.
9. E. Evans, *Domain-Driven Design: tackling complexity in the heart of software*. Addison-Wesley, 2004.
10. M. Aiello, *The Web was done by amateurs*, Springer, 2018.
11. P. J. Houston, "Introduction to DCE and Encina," *Whitepaper, Transarc Corp*, 1996.
12. S. Vinoski, "CORBA: integrating diverse applications within distributed heterogeneous environments," *IEEE Communications mag.*, vol. 35, no. 2, pp. 46–55, 1997.
13. R. Sessions, *COM and DCOM: Microsoft's vision for distributed objects*. John Wiley & Sons, Inc., 1997.
14. D. S. Platt, *Understanding COM+*. Microsoft Press, 1999.
15. G. Alonso, F. Casati, H. A. Kuno, and V. Machiraju, *Web services. Concepts, architectures and applications*. Springer, 2004.
16. R. T. Fielding and R. N. Taylor, "Principled design of the modern Web architecture," *ACM Trans. on Internet Technology*, vol. 2, no. 2, pp. 115–150, 2002.
17. C. Pautasso, O. Zimmermann, and F. Leymann, "Restful Web services vs. "big" Web services: making the right architectural decision," in *Proc. of 17th WWW*, pp. 805–814, 2008.
18. P. Castro, V. Ishakian, V. Muthusamy, and A. Slominski, "The rise of serverless computing," *Communications of the ACM*, vol. 62, no. 12, pp. 44–54, 2019.

19. L. J. Bass, I. M. Weber, and L. Zhu, *DevOps - A software architect's perspective*. SEI series in software engineering, Addison-Wesley, 2015.
20. K. Baina, B. Benatallah, F. Casati, and F. Toumani, "Model-driven Web service development," in *Proc. CAiSE 2004*, pp. 290–306, Springer, 2004.
21. D. Skogan, R. Grønmo, and I. Solheim, "Web service composition in UML," in *Proc. EDOC 2004*, pp. 47–57, IEEE, 2004.
22. I. Graham, *Requirements modelling and specification for Service-Oriented Architecture*. John Wiley & Sons, 2008.
23. N. Milanovic and M. Malek, "Current solutions for Web service composition," *IEEE Internet Computing*, vol. 8, no. 6, pp. 51–59, 2004.
24. A. L. Lemos, F. Daniel, and B. Benatallah, "Web service composition: a survey of techniques and tools," *ACM Computing Surveys (CSUR)*, vol. 48, no. 3, pp. 1–41, 2015.
25. D. Berardi, D. Calvanese, G. De Giacomo, M. Lenzerini, and M. Mecella, "Automatic composition of e-services that export their behavior," in *Proc. ICSOC 2003*, pp. 43–58, Springer, 2003.
26. D. Berardi, D. Calvanese, G. De Giacomo, R. Hull, and M. Mecella, "Automatic composition of transition-based semantic Web services with messaging," in *Proc. 31st VLDB*, pp. 613–624, VLDB, 2005.
27. D. Calvanese, G. De Giacomo, M. Lenzerini, M. Mecella, and F. Patrizi, "Automatic service composition and synthesis: the Roman model," *IEEE Data Eng. Bull.*, vol. 31, no. 3, pp. 18–22, 2008.
28. S. A. McIlraith, T. C. Son, and H. Zeng, "Semantic Web services," *IEEE Intelligent Systems*, vol. 16, no. 2, pp. 46–53, 2001.
29. D. Fensel and C. Bussler, "The Web service modeling framework WSMF," *Electronic Commerce Research and Applications*, vol. 1, no. 2, pp. 113–137, 2002.
30. D. Ryzko, *Modern big data architectures: a multi-agent systems perspective*. John Wiley & Sons, 2020.
31. A. Bouguettaya, M. Singh, M. Huhns, Q. Z. Sheng, H. Dong, Q. Yu, A. G. Neiat, S. Mistry, B. Benatallah, B. Medjahed, *et al.*, "A service computing manifesto: the next 10 years," *Communications of the ACM*, vol. 60, no. 4, pp. 64–72, 2017.
32. G. De Giacomo, X. Oriol, M. Estanol, and E. Teniente, "Linking data and BPMN processes to achieve executable models," in *Proc. CAiSE 2017*, pp. 612–628, Springer, 2017.
33. X. Oriol, G. De Giacomo, M. Estañol, and E. Teniente, "Automatic business process model extension to repair constraint violations," in *Proc. ICSOC 2019*, pp. 102–118, Springer, 2019.