

# Chapter 2

## Approaches to the Evolution of SOA Systems

Norman Wilde, Bilal Gonen, Eman El-Sheikh  
and Alfred Zimmermann

**Abstract** The evolution of Services Oriented Architectures (SOA) presents many challenges due to their complex, dynamic and heterogeneous nature. We describe how SOA design principles can facilitate SOA evolvability and examine several approaches to support SOA evolution. SOA evolution approaches can be classified based on the level of granularity they address, namely, service code level, service interaction level and model level. We also discuss emerging trends, such as microservices and knowledge-based support, which can enhance the evolution of future SOA systems.

### 2.1 Introduction

Early in the history of modern computing it became evident that most of the software developer's work actually takes place after an application's initial delivery. This work came to be known as "software maintenance". Despite its economic importance, in the literature it was usually relegated to a supposedly uninteresting box at the bottom end of the waterfall software development life cycle.

With time, the term "maintenance" became unpopular because it was found that most of the work had little to do with repair, and much to do with the evolution of user needs and of computing environments [1]. As each new need or environment emerges the application must either adapt, be rewritten, or die.

---

N. Wilde

Department of Computer Science, University of West Florida, Pensacola, FL, USA  
e-mail: nwilde@uwf.edu

A. Zimmermann

Faculty of Informatics, Reutlingen University, Reutlingen, Germany  
e-mail: alfred.zimmermann@reutlingen-university.de

B. Gonen

School of Information Technology, University of Cincinnati, Cincinnati, OH, USA

E. El-Sheikh (✉)

Center for Cybersecurity, University of West Florida, Pensacola, FL, USA  
e-mail: eelsheikh@uwf.edu

So today we speak more of “software evolution” than “software maintenance”. This term is also somewhat problematic since “to evolve” is a passive verb, and thus gives the impression that evolution is something that just happens. In fact, keeping an application up to date requires very hard work, often under cruel deadline pressure, performed by very highly qualified professionals. In this chapter we use both terms since either, “evolution” or “maintenance”, allows us to distinguish a greenfield software development situation in which design decisions can be taken freely, from the highly constrained context faced in making changes to an existing system.

The defining characteristic of software maintenance/evolution as opposed to development is that any proposed change needs to take into account a large base of existing software. This software has usually been molded by decisions taken, possibly years earlier, in circumstances very different from the current reality. Any change is thus highly constrained.

The emergence of Services Oriented Architecture (SOA) systems in the first decade of this century certainly did not eliminate the problems of software evolution, but it did change their nature. As a series of authors have pointed out, some aspects of SOA favor the job of the maintainer while others make it more difficult. New challenges are created both for practitioners and for researchers [2–6].

In this chapter we first briefly discuss perspectives on software evolution in general before going on to highlight some of the main approaches when these perspectives are applied to SOA systems. We cannot attempt to identify all of the diverse approaches to our subject, but we aim to contrast some of the main themes and explore advantages and disadvantages. The books and papers we mention are by no means an exhaustive list, but rather typify different ways of looking at the SOA evolution problem. We close with some discussion of emerging trends both in SOA architectures and in using knowledge-based methods to understand these architectures.

## 2.2 Perspectives on Software Evolution

One can identify two broad perspectives on software evolution that have dominated both theory and practice. On the one hand, software can be designed initially to make evolution easier. This is a perspective for the original software developer. It focuses on design approaches and implementation architectures that are hoped to facilitate future evolution. We might call this approach *design for evolvability*.

The second perspective looks for tools and methods to support ongoing maintenance of an existing system. This is a perspective for a maintenance software engineer. He must accept the system as it is, warts and all, and try to do the best possible job of keeping it up to date. We could call this perspective *support for evolution*.

### 2.2.1 *Design for Evolvability*

In design for evolvability, a key theme has been to find the “right” modularity. Any large application must be implemented as modules that connect together to provide the overall system functionality. The choice of modules, their interfaces, and the connection methods all strongly affect the ease with which changes can be made.

A key initial insight was the concept of “information hiding”, generally credited to Parnas writing in 1972:

We propose instead that one begins with a list of difficult design decisions or design decisions which are likely to change. Each module is then designed to hide such a decision from the others [7].

A designer thus should anticipate change, and hide expected changes within a single module. If the maintainer has to make one of the expected changes, then he or she can work within that single module. Design, coding and testing of the change will thus be far simpler than they would have been if many modules had been affected.

The difficulty for the designer, of course, is in identifying the likely changes so early in the system’s lifetime. Security and performance considerations can also constrain what data and functionality need to be kept together, and thus lead to a decomposition that may seem less than optimal.

It is also not trivial to find a way of implementing the resulting modules without encountering insuperable barriers in the programming language or runtime environment. Many of the advances of programming languages have involved providing better mechanisms for modularity, with an explicit or implied goal of facilitating information hiding.

### 2.2.2 *Support for Evolution*

A great diversity of tools has been proposed to support the evolution of an existing software application. One might think of configuration management systems, editors that compare software versions, impact analysis tools, regression testing frameworks, and so on. But one important theme has been to provide the maintainer with support for *program comprehension*.

If the biggest difference between development and evolution is the presence of an existing code base, then the biggest practical difference to the software engineer is his need to *understand* that code base. The developer presumably understands the code he deals with because he, or his immediate colleagues, wrote it. The maintainer is often separated from the code’s original authors by a distance of many miles or many years. He or she must reconstruct sufficient understanding of the application to be able to change it safely, without unexpected and possibly disastrous side effects.

Understanding legacy software is a complex task, both because of the scale of many existing software applications and because of the variety of relationships that may need to be understood. It has long been clear that software maintainers cannot attempt to understand large applications in their entirety or each maintenance task would take far too long. Instead they try to use a pragmatic as-needed strategy to understand only what is immediately relevant for the task at hand [8]. Even within that limitation, however, there is a bewildering variety of information which may be relevant: the structure of program text, dynamic structure and control flow at runtime, program functionality and programming plans at different levels of abstraction, domain knowledge which relates concepts in the real world to structures in the code, and so on [9].

### 2.3 Design for Evolvability Approaches to SOA

Services Oriented Architecture is generally regarded not as a specific architecture, but rather as a general architectural style for structuring software applications. Terminology varies but typically *composite applications* are constructed by orchestrating *services* running on different *nodes* and communicating via message passing. Often an infrastructure layer, sometimes called an *Enterprise Service Bus* (ESB), mediates service interactions providing functions such as message routing, reliable messaging and data transformations (Fig. 2.1).

Within the broad constraints of this style, there are many different ways of architecting any particular application. Many commentators on SOA have enunciated sets of design principles to guide this process (e.g. [10]), and many of the principles have evolvability as a goal. Some of the principles have become common practice while others are still somewhat aspirational.

One general principle is *loose coupling* of services, meaning generally that the designer tries to reduce dependencies between services as much as is practicable. Loose coupling may aid evolution because changes that would otherwise have required the intervention of a maintenance software engineer are instead handled

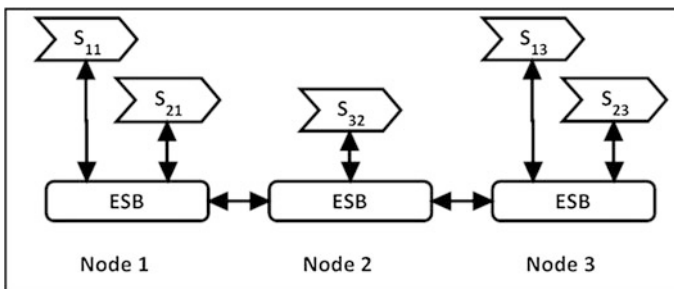


Fig. 2.1 Structure of a simple SOA composite application

automatically. For example at run time each service can publish its address in a registry, so that other services can find it automatically. The alternative of a hardwired address in code would require an edit, a build, and a new deployment. However as Josuttis points out, many of the more advanced strategies designed to provide loose coupling, such as asynchronous communication and error handling by compensation, have the side effect of increasing the complexity of code and thus may hinder maintainability [11].

Perhaps the most universally applied SOA design principle is that each service should implement a published interface or contract. This principle restates the module information hiding principle mentioned earlier; the service is a module that hides all the implementation details behind the interface. The Web Services Description Language (WSDL) [12] was developed to standardize interface descriptions across different hardware and software platforms. If the interface is unchanged, the service implementation can evolve with little or no impact on other services or on the application as a whole.

The WSDL standard greatly aids runtime linking of services, but it still leaves great flexibility in the design of services and their interfaces. These design decisions can have a great impact on evolvability. For example Borovski et al. [13] argue that generic services with flexible interfaces should often be preferred. However there is a tradeoff since a generic interface is less explicit about the data a service expects to receive, and thus provides less guidance to service consumers.

An alternative to formulating general principles for SOA design is to formulate design *patterns* to guide SOA design. This approach recognizes the difficulty of establishing principles that are universally applicable and instead defines patterns that have been found to be effective to achieve specific design goals in specific circumstances. These design goals often have to do with evolvability and specifically with managing versioning of a service as it changes (see [14], especially Chap. 7).

WSDL interface descriptions also aid evolution by facilitating interoperability of components from diverse owners. Interoperation has been described as having two or more independent systems operate in a coordinated and meaningful fashion such that processes are effectively merged or information is effectively shared [15]. If reusable, interoperable components are utilized, then both development and maintenance of the composite application should be easier because less special-purpose coding should be required.

However service reuse, as an organizational SOA strategy, seems to have been more difficult to achieve. Some companies have set out to build portfolios of reusable services with the idea that these will then be composed into new applications as the need arises. In some cases the concept seems to be that business process modeling tools will allow this composition to be done by business experts with little intervention from scarce software engineers. However the development of large-scale reusable software components has always been difficult [16] and the complexities of the assembly of distributed components are daunting. It is perhaps not surprising that Josuttis finds that assembly is best done "... by business and IT experts sitting together" and that service reuse is often less than expected [11]. In

general the contribution of reuse to SOA evolution has probably been much less than originally anticipated.

## 2.4 Support for Evolution Approaches to SOA

As previously stated, the support for evolution perspective looks for tools and methods to aid in the ongoing maintenance of an existing system. The maintainer has to take the system as it is and solve problems effectively while working within organizational time constraints. A key theme in this perspective is helping the maintainer understand the existing system so that he or she can make changes safely.

As we look at SOA composite applications, perhaps one way to classify the different support for evolution approaches would be to look at the level of granularity they address. We could distinguish:

1. Service code level approaches that focus on understanding and manipulating the source code for a service.
2. Service interaction level approaches that focus on understanding how services work together in the application.
3. Model level approaches that focus on how the services relate to models of the application's domain.

### 2.4.1 *Code Level Approaches to SOA Evolution*

Historically, many organizations decided to begin their SOA efforts by exposing existing data or functionality as web services. Vendors soon moved to provide tools to automate this process so that it became easy to expose code written in a wide range of languages, from COBOL to C#, and hosted on platforms ranging from mainframes to Linux<sup>™</sup>.

The tools vary in their capabilities, but it is common to provide the ability to take existing code and create a service and its WSDL, or to take a WSDL and create shell code for a client or a service implementation. For example in the Java environment, one can take a Java class annotated with `@WebService` and use the *wsgen* tool to create the WSDL and classes that will handle the messaging. Going the other way, one can take a WSDL from an existing service and use the *wsimport* tool to create shell code for a client to access that service. Finally, if using a “WSDL-first” or “contract-first” development style, one may create the WSDL by hand and, once it is approved by all stakeholders, use it to generate shell code for the service implementation [17].

If a composite application was developed using source code based tools, then it is natural to continue maintaining it using these same tools. One advantage is that

most of the vendor tools are available through an integrated development environment (IDE) that supports program comprehension with code search, code navigation and debugging facilities.

However a focus on code and code-generating tools for SOA evolution also has several pitfalls. If used incautiously, the tools can generate unwanted dependencies between a service and its client, thus tightening the coupling between them. To take just one example, a software engineer may accidentally include in an interface some of a service's internal data types. Then the client will necessarily have to use these same data types. Any change to the data structure on the server will force a change in the client. The more such code-generating tools are used over the life of a system, the more likely it is that this sort of design flaw will be introduced.

An important consideration may be the *depth* of the change. Papazoglou et al. [18] distinguish between shallow changes, that affect a single service and its immediate clients, and deep changes whose effects may cascade widely within a system. Perhaps a code focus may be acceptable in dealing with shallow changes, but deep changes require a more complex service life cycle to allow for more complete analysis and for more time for changes to propagate across the service landscape.

#### ***2.4.2 Service Interaction Level Approaches to SOA Evolution***

Many of the tasks involved in SOA evolution do not require studying the code, but rather focus on understanding the interactions between services. For example a maintainer may be considering reconfiguring or replacing a service. Or he may need to locate where particular kinds of data are exchanged or where performance bottlenecks are developing. Much of the research on SOA maintenance and program comprehension has thus focused at the service interaction level of granularity.

It may be convenient to distinguish here between tools implementing dynamic and static approaches, since the practicalities of using tools will be different in each case. Dynamic tools get their input from actual execution of the system and use logs or message traces, often supplemented to meet the needs of the tool. Static approaches take as their input descriptions of the SOA system, such as requirements or design documentation, UML models, WSDL interface descriptions, etc. Each approach has its advantages and drawbacks.

One of the earliest dynamic approaches came from a group at IBM. De Pauw et al. [19] describe a visualization tool, Web Services Navigator, which helps users to understand SOA applications better. The tool collects data from event logs and processes it to generate visual abstractions, such as flow patterns, as well as views of transaction flows and data content. The paper describes how the tool has been used to understand overall application behavior in several different problem solving scenarios.

A narrower application of dynamic analysis attempts to recover and understand feature sequences, that is, the service interaction messages that occur when an end user makes use of a particular feature of the application. The problem is that there may be other concurrent users or routine system interactions that obscure the desired feature. Coffey et al. compute a relevance index for each observed message, giving greater weight to messages that are seen when a feature is known to be active. A Feature Sequence Viewer lets the maintainer set a threshold to view the sequence of the most relevant messages [20].

Zawawy et al. [21] present an interesting method that combines dynamic analysis of service logs with preliminary manual encoding of requirements information in goal trees. Their objective is to aid in root cause analysis of failures during corrective maintenance. Their method compares the events recorded in the logs with the goal trees describing expected behavior to locate the fault that is the root cause for a failure.

Chen et al. [22] describe a general framework that can be used to collect dynamic information to monitor SOA applications for a wide variety of maintenance and evolution tasks. They aim to integrate monitoring techniques into web service frameworks, so that the information for dynamic analysis will be transparently available for all applications using the framework.

Espinha et al. also use dynamic analysis to describe the runtime topology of a SOA application, by which they mean identifying which services are running, and how they depend and interact with one another. They provide an interesting analysis of the data a maintainer needs for different evolution scenarios. Their *Serviz* tool intercepts incoming requests to each service to capture the data they need for system visualizations [23].

The dynamic analysis approaches have many advantages. As can be seen from the examples we have cited, dynamic data can support striking visualizations to provide insight into the running system. Also dynamic data comes from the as-built system and thus, unlike models or documentation, is reliably up to date. However it can be difficult to take data from a running system at all the necessary points without encountering instrumentation, performance, and even confidentiality difficulties. Also, any dynamic data depends on what the system was doing at the moment when it was being observed; exceptional or rare behavior may be missed.

Static analysis methods, on the other hand, try to help a maintainer understand a SOA application without having to run it. They thus avoid the data collection problems of dynamic analysis, but with some costs.

One simple approach is to build on well-established search technologies to support SOA maintainers. The *SOAMiner* tool searches both text documentation and WSDLs, XML data schemas, and Business Process Execution Language (BPEL) code [24]. As well as conventional search, the tool also has a rule-based SOA Intel component that creates searchable abstractions from any XML structured inputs. The abstractions summarize relationships within the system and were defined based on the results of SOA comprehension case studies [25].

The problem with the search-based approach is that it largely leaves it up to the maintainer to formulate queries and understand the responses. The abstractions extracted from WSDLs, data descriptions, and BPEL can go only so far in



providing high-level understanding of the system. However to go beyond the search-based approach seems to require additional inputs which may or may not be present for an existing SOA application.

For example Coffey et al. reverse engineer the WSDLs, data descriptions and BPELs into automatically generated concept maps which provide a convenient visualization of the SOA system. The intent is to then conduct interviews with system experts to annotate these maps into a more complete system description that would document it for future maintenance [26].

In another static approach, Kabzeva et al. present a very interesting method that focuses on managing the relationships in a large service network. However as well as the WSDL and BPEL inputs mentioned previously, they require information in modeling notations such as Business Process Model and Notation (BPMN) and Event-Driven Process Chains (EPC). If this information is present they can provide a relationships model that would seem to be very useful for maintenance tasks such as impact analysis [27].

Similarly Bauer et al. propose the use of a SOA repository with advanced analysis capabilities to identify relationships between the services and perform several important kinds of analysis, both to detect already existing problems (as-is-analyses), as well as problems that might occur due to future service changes (what-if-analyses). However they do not make clear how their repository would be populated with information and what manual inputs may be required [28].

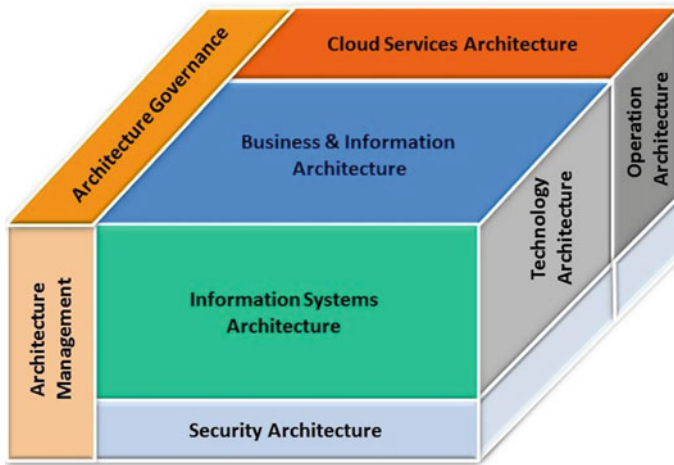
So for static analysis at the service interaction level, there is a tradeoff between undemanding approaches such as search that leave a great deal to the user, and more sophisticated approaches with more complete results, but that require human data collection or accurate pre-existing system models.

### ***2.4.3 Model Level Approaches to SOA Evolution***

Modeling approaches can be used to support SOA evolution. Such approaches focus on the development of models to represent service oriented or enterprise architectures and the use of such models to guide their maintenance and evolution.

One such example is the OASIS Reference Model for Service Oriented Architecture [29], which is an abstract framework that guides reference architectures [30]. The ESARC—Enterprise Services Architecture Reference Cube [31] (Fig. 2.2) is more specific and completes these architectural standards in the context of EAM—Enterprise Architecture Management, and extends these architecture standards for services and cloud computing.

ESARC provides an abstract model to support the integration of business architectures with application architectures and implementation of service-based enterprise systems, and with the technology and operation architecture. ESARC is an original architecture reference model, which provides an integral view for main interweaved architecture types. ESARC abstracts from a concrete business scenario or technologies. The Open Group Architecture Framework provides the basic



**Fig. 2.2** ESARC—enterprise software architecture reference cube

blueprint and structure for the extended service-oriented enterprise software architecture domains like: Architecture Governance, Architecture Management, Business and Information Architecture, Information Systems Architecture, Technology Architecture, Operation Architecture, and Cloud Services Architecture. ESARC provides a coherent aid for the evolution of architectures by facilitating their examination, comparison, classification, quality evaluation and optimization of architectures.

Enterprise Architecture Management for Services Computing is a commonly preferred approach to organize, build and utilize distributed capabilities for Digital Transformation [32]. They provide flexibility and agility in business and IT systems. The development of such applications integrates the Internet of Things (IoT) [33], Web Services [34], REST Services [35], Cloud Computing [36] and Big Data [37], among other frameworks and methods, like architectural semantic support. Today's information systems span a broad range of domains including: intelligent mobility systems and services, intelligent energy support systems, smart personal health-care systems and services, intelligent transportation and logistics services, smart environmental systems and services, intelligent systems and software engineering, intelligent engineering and manufacturing. Microservices [38–40] and the Internet of Things [33] are examples of base technologies for the fast performing digital transformation. The Internet of Things enables a large number of physical devices to connect each other to perform wireless data communication and interaction using the Internet as a global communication environment.

Research reported in [41] focuses on extending the Enterprise Services Architecture Reference Cube (ESARC) by mechanisms for architectural integration and evolution to support adaptable information systems and architectural transformations for changing architectural models. ESARC is an extendable classification framework, which sets a conceptual baseline for digital architectural models.

ESARC makes it possible to verify, define and track the improvement path of different business and IT changes considering alternative business operating models, business functions and business processes, enterprise services and systems, their architectures and related technologies.

To integrate a huge amount of dynamically growing architectural descriptions for services, microservices, or the Internet of Things into consistent enterprise architectures is a considerable challenge. Further research focuses on integrating small EA descriptions for each relevant IoT object. EA-IoT-Mini-Description consists of partial EA-IoT-Data, partial EA-IoT-Models, and partial EA-IoT-Metamodels associated with main IoT objects like IoT-Resource, IoT-Device, and IoT-Software-Component [33]. This research addresses questions such as how can we federate these EA-IoT-Mini-Descriptions to a global EA model and information base by promoting a mixed automatic and collaborative decision process [42]. For the automatic part, model federation and transformation approaches are extended by introducing semantic-supported architectural representations, e.g. by using partial and federated ontologies and associated mapping rules—as universal enterprise architectural knowledge representation, which are combined with special inference mechanisms [43–46].

Metamodels can be used to define architecture model elements and their relationships within ESARC. These metamodels serve as an abstraction for architectural elements and relate them to architecture ontologies [47]. The OASIS Reference Model for SOA [29] is an abstract framework, which defines generic elements and their relationships for service-oriented architectures. Models and metamodels such as ESARC allow software maintainers to navigate the multidimensional space of service oriented and enterprise architectures and facilitate the development and use of semantic-supported navigation and intelligent inferences.

For years semantic technologies were said to revolutionize the web but for the time being the adoption rate is rather low. The basic idea of semantic web is to make the content of the web understandable for machines via the creation of semantic knowledge bases called ontologies. Semantic Web Services are typically extensions to conventional web services [48]. Semantic web services add extra semantic information in order to support automatic web service discovery, automatic web service invocation, automatic web service composition and interoperability [49]. Model Driven Architecture (MDA) uses UML as its preferred modeling language. Semantic models are extremely expressive when modeling structural knowledge. This facilitates modeling as well as maintenance of a model.

Salhofer et al. [50] presents an approach to apply the principles of Model Driven Architecture (MDA) combined with a semantic model. Model Driven Architecture focuses on the creation of models that should be turned into code automatically by code generators. The core idea is to create a model of a system that only represents its functionality but is not influenced by any technological platform. This model is called the Platform Independent Model (PIM). From the PIM, the Platform Specific Model (PSM) is generated. The PSM is then turned into source code by a code generator.

## 2.5 Emerging Trends

This section describes several emerging trends that can support the evolution of SOA systems.

### 2.5.1 *Microservices and Design for Evolvability*

A recent trend in many software application domains has been the shortening of software product delivery cycles. Companies have recognized that there is a strong commercial advantage to providing new features to customers ahead of their competitors. Software is often now delivered as a web application, perhaps combined with a client “app” automatically pushed to customer smartphones. In this environment the customer gets each new version transparently and there is no barrier to releasing new software daily or hourly. Terms used to describe this new software production model include *continuous deployment*, *continuous delivery* and *DevOps* since the roles of software developers and IT system operators become merged [51].

To support this model, new software engineering practices are being adopted such as small teams, tight communication between developers and other stakeholders, identical development and production environments, automatic build on commit, automatic testing on build, and automatic deployment on successful test. For these practices to work, the architecture of the application needs to be carefully planned.

*Microservices* is a name that has been given to an architectural style intended to work within these foreshortened delivery cycles [40]. The term is still relatively new and there is controversy about exactly what constitutes a microservices architecture. Recently at the 11th SEI Architecture Technology User Network (SATURN) Conference, a workshop characterized microservices as shown in Fig. 2.3 [38, 39].

The concept is that there will be small teams each responsible for a few small independent services. Teams will work at their own pace deploying new versions of services when they are ready, without having to coordinate versions. In our terminology, this is clearly a “design for evolvability” approach to SOA evolution. It remains to be seen if this approach will stand the test of time.

### 2.5.2 *Knowledge-Based Support*

Knowledge-based methods can support the evolution of increasingly complex SOA systems of the future. Such methods involve the use of knowledge representations to model SOA systems and reasoning strategies to support maintenance tasks and

**Fig. 2.3** The microservices architectural style

The SOA Architectural style, roughly consistent of these constraints:

- Communication via messages
- Each service is independently deliverable
- Loosely coupled

Plus organizational constraints

- Decentralized design authority
- Architect is a coach
- Architecture is enforced through tooling
- Limited team size

In order to

- Sustain high delivery velocity by removing contention between teams and allowing rapid evolution (i.e. business agility and responding to change)

code comprehension. In particular, ontological modeling can support SOA evolution by representing both high-level business and architectural views of the whole application as well as lower level, code-focused views.

A commonly used ontology, the Open Group SOA ontology, can be extended to develop a *SOA Evolution Ontology* that better addresses software maintenance demands [30]. The Open Group's ontology describes business processes, services and their interfaces in a fairly abstract manner. The maintainer needs that description, but also needs to deal with concrete implementation details as may be found in design rationale, detailed interface specifications and in code. As an example of this approach, an ontology was developed to support semantic browsing and help a maintainer quickly acquire the information needed for a particular maintenance task [52]. A specialized semantic browser can be used to support the navigation of the large repositories of textual, semi-structured artifacts describing a SOA system. Textual artifacts include natural language design rationale, design and code documentation, semi-formal service interface specifications (e.g. WSDLs), BPEL orchestration code, etc. These artifacts are annotated through semantic labels that support discovery of the semantic relations between different artifacts.

Although little research has been reported on the development and use of knowledge-based methods for the maintenance and evolution of SOA systems there is literature on the application of semantic web techniques for maintaining traditional (non-SOA) software systems. The research reported in [52] focused on providing ontological support for software artifacts such as source code and documentation. In work reported by Witte et al. [53], customized ontologies were populated automatically from source code and documentation, and then queried to

provide support for source code security analysis, for traceability links between source code and documentation and for architecture analysis. In work by Hyland-Wood et al. [54], an ontology was developed to describe the relationship between object-oriented software components.

Rastgoo et al. [55] is one of the few papers that take a knowledge-based approach to facilitate software engineering processes for SOA. The paper proposes automated generation of requirements ontologies using UML diagrams. The generated ontology considers the behavior and hierarchical relationship of services. Experimental results demonstrate the improvement of the proposed approach from perspectives, such as completeness and automatic generation of requirements ontology for SOA systems.

Knowledge-based support and ontological models can help address SOA evolution challenges to keep them in continuous service in the face of rapidly changing environments, continually emerging security risks, and a dynamic mix of partner organizations. Future trends will see the development of *ecosystems of ontologies* to describe increasingly complex SOA systems [41]. Consistent modeling approaches will need to emerge to bridge architectural levels and address the different concerns of business experts, developers and maintainers. The task of supporting the evolution of SOA systems will always be challenging, but such knowledge-based models could greatly ease the burden on software maintainers.

## 2.6 Concluding Remarks

In this chapter we described the challenges of software maintenance and evolution and examined various approaches specifically within the context of Services Oriented Architectures. We argue that SOA design principles such as loose coupling and service interfaces can facilitate SOA system evolvability. We described various approaches for SOA evolution support, which were classified by their level of granularity: service code level, service interaction level and model level approaches. We also presented emerging trends in supporting the maintenance and evolution of SOA systems, including microservices and knowledge-based support. Approaches such as the ones examined in this chapter can enhance the evolution of future SOA systems.

## References

1. Lientz, B.P., Swanson, E.B., Tompkins, G.E.: Characteristics of application software maintenance. *Commun. ACM* **21**(6), 466–471 (1978)
2. Gold, N., Mohan, A., Knight, C., Munro, M.: Understanding service-oriented software. *Softw. IEEE* **21**(2), 71–77 (2004)
3. CanforaHarman, G., Di Penta, M.: New frontiers of reverse engineering. *Future Softw. Eng.* (2007) (IEEE Computer Society)

4. Lewis, G., Smith, D.B.: Service-oriented architecture and its implications for software maintenance and evolution. *Frontiers Softw. Maint. (FoSM)* (2008) (IEEE)
5. Kontogiannis, K.: Challenges and opportunities related to the design, deployment and, operation of Web Services. *Front. Softw. Maint. (FoSM)* (2008) (IEEE)
6. Lewis, G.A., Smith, D.B., Kontogiannis, K.: Proceedings of the Fourth International Workshop on a Research Agenda for Maintenance and Evolution of Service-Oriented Systems (MESOA 2010) (2011)
7. Parnas, D.L.: On the criteria to be used in decomposing systems into modules. *Commun. ACM* **15**(12), 1053–1058 (1972)
8. Koenemann, J., Robertson, S.P.: Expert problem solving strategies for program comprehension. In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM (1991)
9. Von Mayrhauser, A.: Program comprehension during software maintenance and evolution. *Computer* **28**(8), 44–55 (1995)
10. Erl, T.: *SOA Principles of Service Design*, vol. 37, pp. 71–75. Prentice Hall, Boston (2007)
11. Josuttis, N.M.: *SOA in Practice: The Art of Distributed System Design*. O'Reilly (2007). ISBN 0-596-52955-4
12. Christensen, E., Curbera, F., Meredith, G., Weerawarana, S.: *Web services description language (WSDL) 1.1* (2001)
13. Borovskiy, V., Mueller, J., Schapranow, M., Zeier, A.: Ensuring service backwards compatibility with generic web services. In: *Proceedings of the 2009 ICSE Workshop on Principles of Engineering Service Oriented Systems*. IEEE Computer Society (2009)
14. Daigneau, R.: *Service Design Patterns: Fundamental Design Solutions for SOAP/WSDL and Restful Web Services*. Addison-Wesley (2011)
15. Scholl, H.J., Klischewski, R.: E-government integration and interoperability: framing the research agenda. *Int. J. Publ. Adm.* **30**(8–9), 889–920 (2007)
16. Glass, R.L.: *Facts and Fallacies of Software Engineering*. Addison Wesley (2003)
17. Hewitt, E.: *Java SOA Cookbook*. O'Reilly Media Inc. (2009)
18. Papazoglou, M.P., Andrikopoulos, V., Benbernou, S.: Managing evolving services. *Softw. IEEE* **28**(3), 49–55 (2011)
19. De Pauw, W., Lei, M., Pring, E., Villard, L., Arnold, M., Morar, J.F.: Web services navigator: visualizing the execution of web services. *IBM Syst. J.* **44**(4), 821–845 (2005)
20. Coffey, J., White, L., Wilde, N., Simmons, S.: Locating software features in a SOA composite application. In: *2010 IEEE 8th European Conference on Web Services (ECOWS)*. IEEE (2010)
21. Zawawy, H., Mylopoulos, J., Mankovskii, S.: Requirements-driven framework for root cause analysis in SOA environments. In: *Proceedings of the Fourth International Workshop on a Research Agenda for Maintenance and Evolution of Service-Oriented Systems (MESOA 2010)* (2011)
22. Chen, C., Zaidman, A., Gross, H.: A framework-based runtime monitoring approach for service-oriented software systems. In: *Proceedings of the International Workshop on Quality Assurance for Service-Based Applications*. ACM (2011)
23. Espinha, T., Zaidman, A., Gross, H.G.: Understanding the runtime topology of service-oriented systems. In: *2012 19th Working Conference on Reverse Engineering (WCRE)*, pp. 187–196. IEEE (2012)
24. Wilde, N., Leal, D., Goehring, G., Terry, C.: Enhanced search: an approach to the maintenance of services oriented architectures. In: *Ninth International Conference on Software Engineering Advances (ICSEA 2014)*. Nice, France, 12–16 Oct 2014
25. El-Sheikh, E., Reichherzer, T., White, L., Wilde, N., Coffey, J., Bagui, S., et al.: Towards enhanced program comprehension for service oriented architecture (SOA) systems (2013)
26. Coffey, J.W., Reichherzer, T., Owsnick-Klewe, B., Wilde, N.: Automated concept map generation from service-oriented architecture artifacts, pp. 49–56 (2012)

27. Kabzeva, A., Götze, J., Lottermann, T., Müller, P.: Service relationships management for maintenance and evolution of service networks. In: *The Eighth International Conference on Software Engineering Advances (ICSEA 2013)* (2013)
28. Bauer, T., Buchwald, S., Tiedeken, J., Reichert, M.: A SOA repository with advanced analysis capabilities-improving the maintenance and flexibility of service-oriented applications (2015)
29. MacKenzie, C.M., Laskey, K., McCabe, F., Brown, P.F., Metz, R., Hamilton, B.A.: Reference model for service oriented architecture 1.0, p. 12. OASIS Standard (2006)
30. Open Group: Service-oriented architecture ontology (2010)
31. Zimmermann, A., Buckow, H., Gross, H., Nandico, O.F., Piller, G., Prott, K.: Capability diagnostics of enterprise service architectures using a dedicated software architecture reference model. In: *2011 IEEE International Conference on Services Computing (SCC)*. IEEE (2011)
32. Zimmermann, A., Schmidt, R., Sandkuhl, K., Jugel, D., Moehring, M., Wissotzki, M.: Enterprise architecture management for the internet of things. *Lecture Notes in Informatics* (2015), Dec 15, Boeblingen, Germany
33. Patel, P., Cassou, D.: Enabling high-level application development for the internet of things. *J. Syst. Softw.* **103**, 62–84 (2015)
34. Papazoglou, M.P., *Web Services & SOA: Principles and Technology*. Pearson—Prentice Hall (2012)
35. Ebert, J., Erl, T., Carlyle, B., Pautasso, C., Balasubramanian, R.: SOA with REST: Principles, Patterns & Constraints for Building Enterprise Solutions with REST. *ACM SIGSOFT Software Engineering Notes*, vol. 38(3), pp. 32–33 (2013)
36. Marinescu, D.C.: *Cloud Computing: Theory and Practice*. Newnes (2013)
37. Berman, J.J.: *Principles of Big Data: Preparing, Sharing, and Analyzing Complex Information*. Newnes (2013)
38. Microservices Workshop at SATURN 2015 [Internet]. Available from: <https://saturnnetwork.wordpress.com/2015/05/07/microservices-workshop-at-saturn-2015>
39. SATURN2015-Microservices-Workshop Key Outcomes [Internet]. Available from: <https://github.com/michaelkeeling/SATURN2015-Microservices-Workshop/blob/master/outcomes/key-outcomes.md>
40. Newman, S.: *Building Microservices*. O'Reilly Media, Inc. (2015)
41. Zimmermann, A., Gonen, B., Schmidt, R., El-Sheikh, E., Bagui, S., Wilde, N.: Adaptable enterprise architectures for software evolution of smart life ecosystems. In: *2014 IEEE 18th International Enterprise Distributed Object Computing Conference Workshops and Demonstrations (EDOCW)*. IEEE (2014)
42. Jugel, D., Schweda, C.M., Zimmermann, A.: Modeling decisions for collaborative enterprise architecture engineering. In: *Advanced Information Systems Engineering Workshops*. Springer (2015)
43. Breu, R., Agreiter, B., Farwick, M., Felderer, M., Hafner, M., Innerhofer-Oberperfler, F.: Living models-ten principles for change-driven software engineering. *Int. J. Softw. Inform.* **5** (1–2), 267–290 (2011)
44. Farwick, M., Pasquazzo, W., Breu, R., Schweda, C.M., Voges, K., Hanschke, I.: A meta-model for automated enterprise architecture model maintenance. In: *2012 IEEE 16th International Enterprise Distributed Object Computing Conference (EDOC)*. IEEE (2012)
45. Trojer, T., Farwick, M., Häusler, M., Breu, R.: Living modeling of IT architectures: challenges and solutions. In: *Software, Services, and Systems*, pp. 458–474. Springer (2015)
46. Khan, N.A.: Transformation of enterprise model to enterprise ontology (2011)
47. Zimmermann, A., Zimmermann, G.: Enterprise architecture ontology for services computing. In: *Service Computation*, pp. 64–9 (2012)
48. Alonso, G., Casati, F., Kuno, H., Machiraju, V.: *Web Services*. Springer (2004)
49. Martin, D., Paolucci, M., McIlraith, S., Burstein, M., McDermott, D., McGuinness, D., et al.: Bringing semantics to web services: the OWL-S approach. In: *Semantic Web Services and Web Process Composition*, pp. 26–42. Springer (2005)
50. Salhofer, P., Stadlhofer, B.: Semantic MDA for e-government service development. In: *2012 45th Hawaii International Conference on System Science (HICSS)*. IEEE (2012)



51. Wikipedia: DevOps [Internet] (2015). Available from: <http://en.wikipedia.org/wiki/DevOps>
52. Gonen, B., Fang, X., El-Sheikh, E., Bagui, S., Wilde, N., Zimmermann, A., et al.: Maintaining SOA Systems of the future—how can ontological modeling help? In: KEOD 2014—Proceedings of the International Conference on Knowledge Engineering and Ontology Development, Rome, Italy, 21–24 Oct 2014
53. Witte, R., Zhang, Y., Rilling, J.: Empowering software maintainers with semantic web technologies. In: *The Semantic Web: Research and Applications*, pp. 37–52. Springer (2007)
54. Hyland-Wood, D., Carrington, D., Kaplan, S.: Towards a software maintenance methodology using semantic web techniques and paradigmatic documentation modelling. *Softw. IET* **2**(4), 337–347 (2008)
55. Rastgoo, V., Hosseini, M., Kheirkhah, E.: Semantic web-based software engineering by automated requirements ontology generation in SOA. *Int. J. Web Seman. Technol.* **5**(2), 1 (2014)

## Author Biographies

**Norman Wilde** is Nystul Chair and Professor of Computer Science at the University of West Florida. He received his Ph.D. in Mathematics and Operations Research from the Massachusetts Institute of Technology in 1971. His research interests are Software Engineering, Software Maintenance/Evolution, Services Oriented Architectures and Cybersecurity.

**Bilal Gonen** is an Assistant Professor of Information Technology at the University of Cincinnati. He received his Ph.D. in Computer Science and Engineering from University of Nevada, Reno, in 2011. His research interests are Software Engineering, Services Oriented Architectures, Computer Networks, Complex Networks, Social Network Analysis, Semantic Web, Algorithms, Machine Learning.

**Eman El-Sheikh** is Professor of Computer Science and Director of the Center for Cybersecurity at the University of West Florida. She received her Ph.D. in Computer Science from Michigan State University in 2001. Her research interests include Artificial Intelligence, Machine Learning, Intelligent Systems, Cybersecurity, Software Maintenance and Evolution, and Services Oriented Architectures.

**Alfred Zimmermann** is Professor of Computer Science at Reutlingen University and Research Director of the Herman Hollerith Center for Services Computing Boeblingen, Germany. His research is focused on Digital Transformation and Digital Enterprise Architecture in close relationship with Services and Cloud Computing. He graduated in Medical informatics at the University of Heidelberg and got his Ph.D. in Informatics from the University of Stuttgart, Germany.

Emerging Trends in the Evolution of Service-Oriented  
and Enterprise Architectures

El-Sheikh, E.; Zimmermann, A.; Jain, L.C. (Eds.)

2016, XVI, 265 p. 97 illus., 49 illus. in color., Hardcover

ISBN: 978-3-319-40562-9