Association for Information Systems
**AIS Electronic Library (AISeL)**

AMCIS 2009 Proceedings

Americas Conference on Information Systems (AMCIS)

2009

# Towards a Consistent Service Lifecycle Model in Service Governance

Michael Niemann
*TU Darmstadt*, niemann@kom.tu-darmstadt.de

Michael Appel
*TU Darmstadt*, mappel@rbg.informatik.tu-darmstadt.de

Nicholas Repp
*TU Darmstadt*, repp@kom.tu-darmstadt.de

Ralf Steinmetz
*TU Darmstadt*, steinmetz@kom.tu-darmstadt.de

# Towards a Consistent Service Lifecycle Model in Service Governance

**Michael Niemann**
TU Darmstadt, Germany
niemann@kom.tu-darmstadt.de

**Michael Appel**
TU Darmstadt, Germany
mappel@rbg.informatik.tu-darmstadt.de

**Nicolas Repp**
TU Darmstadt, Germany
repp@kom.tu-darmstadt.de

**Ralf Steinmetz**
TU Darmstadt, Germany
steinmetz@kom.tu-darmstadt.de

## ABSTRACT

Introducing an SOA in a company brings new challenges for the existing management. Small loosely coupled services allow the Enterprise Architecture to flexibly adapt to existing business processes that themselves depend on changing market environments. SOA, however, introduces a new implicit system complexity. Service Governance approaches address this issue by introducing management processes and techniques, and best practices to cope with the new heterogeneity. Service lifecycle management is one aspect. Existing definitions of service lifecycles vary greatly.. In this paper, we compare existing service lifecycle approaches concerning defined phases and process. In particular, we challenge the purpose of the distinctions made between design time, runtime, and change time. Concluding, we propose a consolidated service lifecycle model for usage in Service Governance.

## Keywords

Service lifecycle, Service-oriented Architecture (SOA), Service Governance, runtime, design time, change time

## INTRODUCTION

During the past few years, the Service-oriented Architecture (SOA) paradigm has proved of value in enabling agile business processes due to the use of standards, loose coupling of services, and a dynamic service binding [Papazoglou 2003, Newcomer and Lomow 2004, Huhns and Singh 2005]. The introduction of an SOA, however, challenges the existing IT management. Small loosely coupled services allow the Enterprise Architecture to flexibly adapt to existing business processes that themselves depend on changing market environments. The classical IT Governance is defined to establish regulation of responsibility, authority, and communication to empower people in terms of decision rights [Weill and Ross 2004]. Furthermore, IT Governance is meant to establish mechanisms of measurement, policy standards, and control to enable the empowered people to carry out their responsibilities within the previously defined roles and authority [Clark 2005, Kjaer 2004]. However, these measures do not completely cover the new inbuilt complexity of an SOA. The number of new software artifacts dramatically increases, and the functional range of each service is rather small. These conditions motivate governance structures to regulate, direct, and control the SOA system, in particular targeting complexity reduction and conformity [Marks and Bell 2006, Niemann et al., 2008].

In these circumstances, lifecycle management is considered more crucial than in a governance approach for a monolithic IT system. In order to successfully operate a service-oriented system, services are subject to a multitude of regulations, e.g., service operation policies, technological policies, security policies, design time policies, and many more [Afshar 2007, Marks and Bell 2006, Mukhi and Plebani 2004, Niemann et al. 2008]. Governance regulations directly address service development and usage. Each service passes several phases such as specification and implementation before it is deployed, used, revised if necessary, and finally, replaced. Governance objectives are to provide a traceable and certifiable service development process. Most of the existing lifecycle definitions do not suitable here.

Many different approaches regarding service lifecycles have been developed and used by academia and software companies. Very often, one can find the distinction between *design time, runtime*, and *change time* – each of them covering a number of different lifecycle phases. In this paper, we outline and compare existing lifecycle approaches and, in particular, challenge the purpose of the distinctions made between these three 'times'.

In the first part of this paper, we compare existing approaches in order to identify characteristics and congruencies. In particular, we investigate and challenge the purpose of *change time* within the service lifecycle. Following, we describe related work. Based on these considerations, we present our service lifecycle approach. We outline the way in which each of the presented approaches can be mapped to our service lifecycle approach.

## COMPARISON

We conducted a mapping of ten service lifecycle propositions envisioned by industry and academia to the SE lifecycle. In the following, we discuss these approaches and investigate whether and how the super-phases design time, runtime, and change time are addressed. We perform a matching of all presented lifecycles to the waterfall model (cf. Fig. 2). The white and grey lanes mark the phases corresponding to one of the SE lifecycle phases. State-intersection could not completely be avoided; however, it has been minimized. Dashed lines indicate the super-phases design time, runtime, and change time.

McBride at IBM defines four service lifecycle phases: *model, assemble, deploy,* and *manage* [McBride 2007]. *Model* covers the phases requirements analysis and design. *Assemble* refers to the phase 'implementation and testing'. *Deploy* maps to the phases 'integration' and 'implementation and testing', as it contains the testing of the assembled service and its further deployment. *Manage* is covered by the classical operation phase. No detailed procedures and operations are defined. It does not include a system analysis, but grounds his lifecycle approach on IBM's SOA Governance lifecycle consisting of the same phases.

Woolf at IBM proposes a model consisting of the phases *Planned*, *Test*, *Active*, *Deprecated,* and *Sunsetted* [Woolf 2006]. A service is planned, designed, implemented, and tested, where the phase *Planned* covers design and *Test* covers implementation and testing. During usage (*active*), it can be marked deprecated to inform about future retirement (*sunsetted).* Woolf's approach differs from the classic SE lifecycle, as no cyclic iteration is present and an explicit *deployment* is not considered. Woolf claims that service maintenance occurs rarely. In spite of changing services, rather *new services* are designed and released. Change*,* therefore, is rarely necessary.

Strnadl (2007) proposes a lifecycle consisting of four development phases (*architecture, development, test,* and *production*) comprising twelve sub-phases. *Architecture* covers requirements analysis and design, while *development* and *test* are specified separately. *Production* covers deployment and usage. This lifecycle approach does not cover requirements analysis nor cyclic reiterations. Once developed and deployed, a service cannot be changed again.

Authors at Software AG (2008) present a non-cyclic lifecycle approach, considering the analysis of a service part of portfolio management. After the service is designed, built, tested, and deployed, it is operated covering service monitoring and change. No change time is taken into account. A change phase as well as the ability to reiterate the lifecycle is missing.

The approach by Windley (2006) explicitly distinguishes between the phases *design, implement, deploy* at *design time*, and *manage*, and *retired* at *runtime*. It does not include a requirements or system analysis. As a cyclic approach, it considers service retirement during the phase *manage*, or change by reiterating starting with the phase *design*. However, it is unclear which processes are exactly covered within *manage* – we assumed *operation*. The approach does not consider change time.

Oracle's lifecycle [Wall 2006a, Wall 2006b] is a cyclic approach. A service traverses the *identifying business process* phase before being modeled. *Build and compose* obviously covers implementation. This lifecycle is the only approach that implements a *publish and provision* phase before the service deployment. "Securing" describes the adaption to customer needs. A service is *secure*d *and manage*d after deployment (*integrate and deploy*). The following *evaluation* phase covers service operation. A reiteration of the lifecycle is possible, starting with the first phase. Change time is not included.

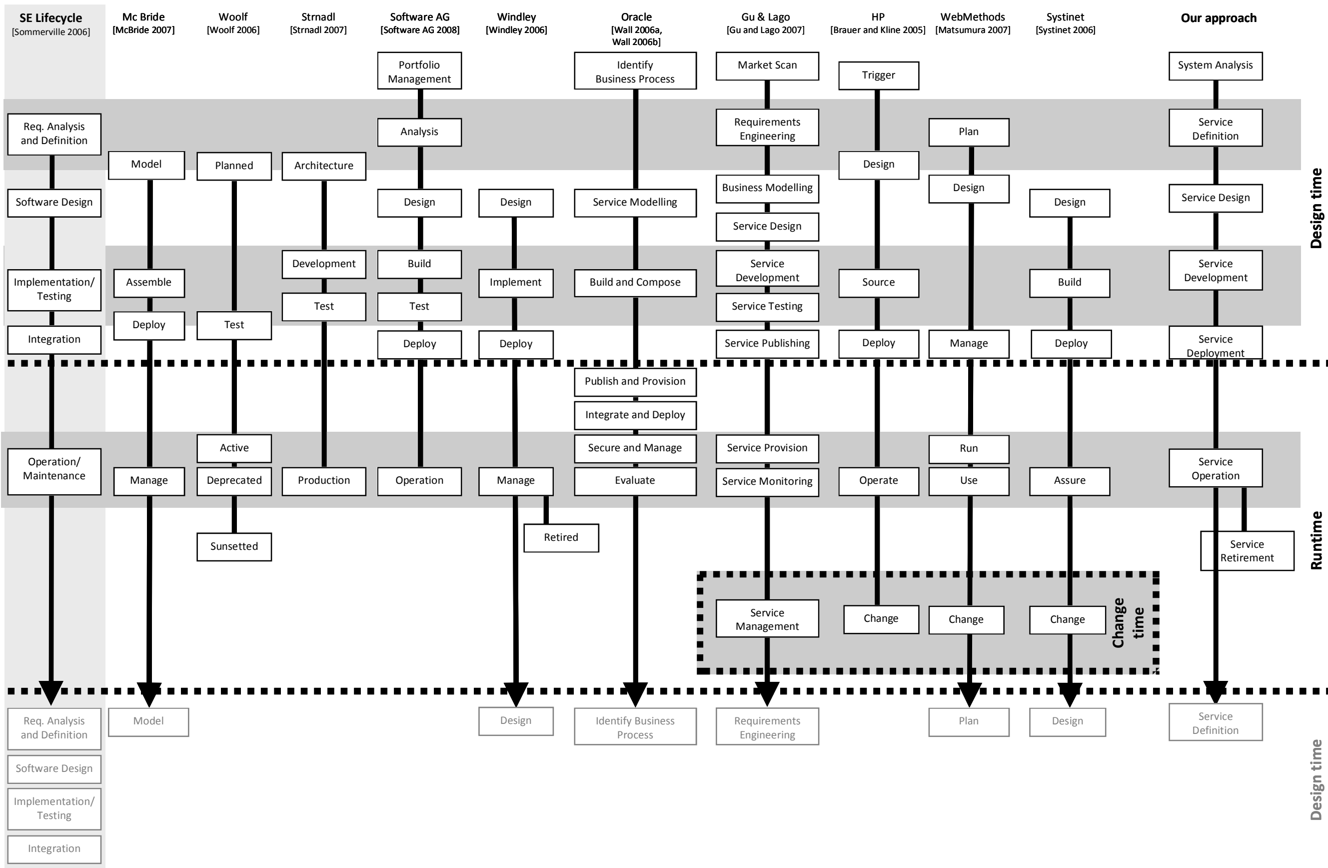


**Figure 2. Service Lifecycle comparison**

The lifecycle by Gu and Lago (2007) is a cyclic approach implementing ten phases, starting with a *market scan*. They outline the service's runtime to begin after a service is published or deployed. This agrees with Sommerville's (2006) classification, while contradicting Oracle's understanding of the order how to publish and integrate a service. The authors explicitly highlight design time and change time as necessary for a well-defined lifecycle approach. According to their approach, a service enters the change time after being provisioned and monitored. Service change takes place before reiterating the lifecycle, entering at *requirements engineering*. The approach does, except for the definition of change time and the total number of phases, match the classical SE lifecycle.

HP proposes a lifecycle with little information about the involved phases [Brauer and Kline 2005]. The *trigger* phase determines the need for a new service and triggers its *design*. The following phases are *design*, covering the requirements analysis and design, and *source*, which covers the implementation and testing, followed by *deploy*, and *operate*. This lifecycle is the only one that implements a never-ending *change* time. A cycle reiteration is not intended. Compared to the SE lifecycle, this change phase maps the reiteration of the complete cycle. Service retirement or a cycle exit is not considered.

Matsumura (2007) at WebMethods proposes a cyclic approach including change time and a change phase. Services are *plan*ned and *design*ed during design time. *Manage* covers deployment, while *run* and *use* correspond to the operation phase. For these phases, the authors lack to provide sufficient descriptions. Following *use*, it enters change time covering, e.g., operational change and version management. After *change*, the cycle can be reiterated.

Authors at Systinet (2006) propose a cyclic lifecycle approach that implements the phases *design, build, deploy*, *assure,* and *change*. While the first three are similar to the SE lifecycle, *assure* corresponds to 'Operation/Maintenance'. *Assur*ing a service can be understood similar to Oracle's *secure and manage* phase. If necessary, service corrections or adjustments are performed during the *change* phase. Additionally, cycle reiteration is considered starting with the *design* phase. The approach does not cover a requirements analysis phase.

The majority of the approaches considers phases that are comparable to the SE lifecycle. The differences originate from different understandings and differing levels of detail. Except for Oracle's lifecycle, design time and runtime cover comparable phases. Only two out of eleven approaches consider leaving the lifecycle for retirement. The main difference, however, is the inclusion of change time. Only four out of eleven investigated approaches explicitly define change time.

## ANALYSIS AND DISCUSSION

Considering the definitions of the outlined approaches, we give short summaries for the super-phases design time, runtime, and change time in the following. Based on this, we discuss the relevance of each of these super-phases.

### *Design Time*

Within the field of SE, design time covers the first four phases of the classical SE lifecycle: requirements analysis, software design, implementation/testing, and integration (cf. Fig. 1 and 2). The investigated approaches particularly emphasize the following aspects of design time. According to Gu and Lago (2007), it covers the service design satisfying the defined functional and non-functional requirements, refined service interfaces, and the style of interaction between services and their clients (asynchronous or synchronous invocation). Gu and Lago include deployment in runtime. Windley (2006) even defines a "deploy time". Most approaches, however, treat deployment as part of design time. Wall (2006a) emphasizes service categorization, modeling methodology, as well as service building and composing concepts as essential parts of design time. At Software AG (2008), design time comprises requirements analysis concerning the services' behavior, performance, quality-of-service (QoS), and security.

Summarizing, design time covers all activities of the software construction process that take place before and including service deployment.

### *Runtime*

According to Sommerville (2006), runtime covers the *operation and maintenance* phase of the SE lifecycle. This includes the use of the software, as well as maintenance and change management. Service and SLA monitoring, reporting service-level agreement (SLA) behavior and the monitoring of other general performance aspects are also part of the runtime phase [Matsumura 2007, Software AG 2008, and Wall 2006b].

To underline the customer's involvement during the usage phase, Windley (2006), Systinet (2006), Matsumura (2007), and Gu and Lago (2007) additionally present a consumer service lifecycle, which consists of the four phases *discover, bind, use,* and *disconnect* [Windley 2006]. *Discover* describes a service, capable of fulfilling the customer's demand, being discovered within the service provider's registry. *Bind* involves contract management by the provider and consumer, as well as the service level agreement handshake. The *use* phase describes the time until *disconnect* in which the service is actually used by the customer. If necessary, the customer lifecycle can be reiterated, starting at the *discover* state [Windley 2006]. Matsumura (2007) define the consumer service lifecycle to regulate transactions between service provider and service consumer. These interactions include broker transaction, data transformation, message queuing, and security handshakes. Authors at Systinet (2006) define the consumer service lifecycle involving the four phases *discover, bind, interact,* and *monitor*. *Discover* describes the provided service being discovered within the provider's registry. *Bind* involves contract management to negotiate the terms of use for a certain service, followed by the phase *interact*, where the service is used. Finally, in the phase *monitor,* details concerning the quality of service performance are collected and reported to the provider.

Concluding, the understanding of runtime basically covers service operation including monitoring. A consumer lifecycle is also part of this phase, covering activities and phases involving the service consumer starting in the service operation phase.

### *Change Time*

Within the field of SE, change time is not explicitly considered. However, considerations for system improvements, error correction [Sommerville 2006], are mandatory.

Gu and Lago (2007) explicitly define a change time phase following the runtime phase. The trigger they mention is business requirements change which directly affects functionality adjustments. They argue that the ability to change services is required to meet user expectations and stay competitive. Nevertheless, they admit that a lifecycle reiteration is potentially necessary at this point in time. According to authors at Systinet (2006), change time covers service version management and service adaptation to newly encountered security requirements. Following change time, a service restarts processing the lifecycle. Matsumura (2007) emphasizes service version upgrades, operational change of services, service decommissioning, and service deprecation as central aspects of change time. In order to plan and design the outlined operational change, a subsequent reiteration of the cycle is required. According to authors at HP [Brauer and Kline 2005], change time is an endless adaptation process following the runtime phase. Change time involves service delivery, service delivery management, and service version management. A reiteration of the proposed lifecycle is not intended. Obviously, authors at Software AG (2007a) consider change time a crucial component of the service lifecycle. However, they fail to state its definition or purpose.

Given that only four out of eleven approaches consider change time, definitions vary considerably. Most of the authors, however, define change as additional phase following the classic operation phase. So according to these definitions, change time basically covers the management of service decommissioning, versioning, and retirement - including handling of functional changes, covering, i.e., service adoption, business process change, service portfolio change, technical requirements change, and security issues.

### Discussion

Seven out of eleven approaches we reviewed include neither a change phase nor change time. Even the classic SE approach does not involve it, although change of software components or systems has always been an important topic in software engineering.

The original SE lifecycle covers software change and versioning, but does not define or require a change time phase in the corresponding lifecycle. Change is considered a sub-process of *Operation and Maintenance* and triggers a reiteration of the cycle. Each iteration of the lifecycle after the first one represents a software change process.

Basically, change time stands for and addresses the need to manage change activities that are required upon change requests originating from necessary functional changes. It implies addressing the change request in order to solve the occurred problem. As soon as the service itself is concerned, its requirements, definition, and design are reanalyzed – in order to finally accomplish code change with implies testing. Obviously, this is equivalent to reiteration of the service lifecycle, as defined by many of the approaches presented. In fact, Boehm (1988) described a spiral software development process that reiterates one cycle while addressing different evolving issues in each iteration. The approach by Brauer and Kline (2005) almost implements this by specifying a change loop.

Given that the operation and maintenance phase of the classic SE lifecycle includes change handling, the definition of a change phase or time seems unwarranted. It is assumed that it can safely be replaced by a cycle reiteration. Additionally to Matsumura (2007), Systinet (2006), and Gu and Lago (2007), this also exceeds the approaches by Woolf (2006), Strnadl (2007), Software AG (2008), and Brauer and Kline (2005), whose lifecycle does not reflect a cyclic behavior.

After sifting through and analyzing all these arguments, the authors of this paper cannot identify a clear motivation for a change time in a service lifecycle. We systematically disproved all arguments or advantages of such a super-phase. The imperative conclusion is the absence of the need to define and integrate a change time into a service lifecycle.

After a discussion of related work, we define a service lifecycle that allows for the above considerations and extends the presented approaches while forming a generally applicable lifecycle.

## RELATED WORK

Lifecycle processes originate from software engineering (SE). SE processes define any software’s lifecycle from requirement analysis to usage and maintenance. The main purpose is to ensure a structured software development process in a well-defined and cost effective way. [Sommerville 2006, Bhushan and Tayal 2007] There are a number of approaches to software development, i.e. agile methods, extreme developing methods, the “common” waterfall approach, etc. [Sommerville 2006]. Due to linearity and similarity to the waterfall model of all investigated approaches, we use this model as reference.

Service development addresses the creation of very small software artifacts in comparison with today's software development projects – hence, it is easier to react to changes in requirements later on; iterations are less expensive. For this reason this method is still chosen for service development. *Agile software development* methods, for example, are not appropriate, if certain goals are to be completed within a defined timeline or budget, which is often the case in service development. In *extreme software development* methods, requirements are defined incrementally, rather than in advance. Design activities take place incrementally, adding complexity only when failing tests require it. In small development projects like service production, requirements must be defined beforehand due to development procedures, requirements concerning service test case generation, data processing, later traceability, etc. as well as a limited budget is mostly given. Hence, a stringent development process is normally requested by service governance procedures. Service development needs a well-structured, traceable, certifiable process to guarantee consistent and rules compliant service development (and their later execution).

As a reference, we use the typical lifecycle (the "waterfall model"), which consists of the phases *requirement analysis and definition*, s*oftware design*, *implementation and testing, integration,* and *operation and maintenance* [Boehm 2007, Bhushan and Tayal 2007, Jacobson and Bylund 2000, Sommerville 2006]. These cover in detail:

- *Phase 1 - Requirements analysis and definition* defines the software system's purpose, constraints and goals in cooperation with the system users. These aspects compose the system specification.
- *Phase 2 - Software design* distinguishes between hardware and software requirements and defines a general architecture. It covers the identification and description of software components and their relationships.
- *Phase 3 - Implementation and testing* comprises the realization of the defined design. It involves testing to ensure that the software meets the agreed specifications.
- *Phase 4 - Integration* includes system tests to ensure that interaction with other software components works faultlessly. Furthermore within this phase, the software is delivered to the customer.
- *Phase 5 - Operation and maintenance* usually is the longest phase. It includes software usage and monitoring, as well as error-correction processes. Should it become necessary to implement new requirements, a change process is triggered in this phase.

These phases are designed to form a loop. As soon as changes in functionality or requirements are registered, the phase *operation and maintenance* is interrupted and the lifecycle continues with the first phase, *requirements analysis*, or any other phase of the lifecycle, that can directly or better address the occurred problem [Sommerville 2006]. (cf. Fig. 1).

Riedl et al. (2009) perform a service lifecycle comparison of approaches taken from various research areas. In their concluding integrated lifecycle model, they identify the phases *analysis and design, negotiation, provisioning, usage* covering operation, change, and monitoring, and *withdrawal*.

Requirements analysis and design phases are combined in one phase *analysis and design*. The authors combine all activities "preparing the actual operation of a service" in the phase *provisioning*, mentioning that this phase can even include a complete software development lifecycle. During the comparison, all model, implementation, and deployment activities are assigned to this phase. The *negotiation* phase, which is located *before* the *provisioning* phase in the lifecycle, covers contract negotiation, service describing, offering, publishing, and advertising. The content of this phase is not explicitly justified. However, from the ordering of these phases concludes that service offering and publishing is *followed by* service implementation. This segmentation might not apply to all realized and present service lifecycles. Furthermore, in general a service *definition* phase is considered very important concerning services. In this phase, decisions on overall granularity, functionality scope, and interfaces are made. It is considered crucial for a service's ability to be reused [Fabini 2007]. Riedl at al. do not consider such a phase.

As illustrated by these considerations, Riedl et al. discuss the service lifecycle on a higher level of abstraction than we do. For governance-related issues, the (low level) service development process is especially important. We evaluate and discuss lifecycles against the background of governance issues as process certification or traceability. Our intention is to identify a generally applicable lifecycle for usage as reference process that can be used to guarantee conformity during service development. Concluding, the comparison by Riedl has been performed against a different background targeting different objectives.

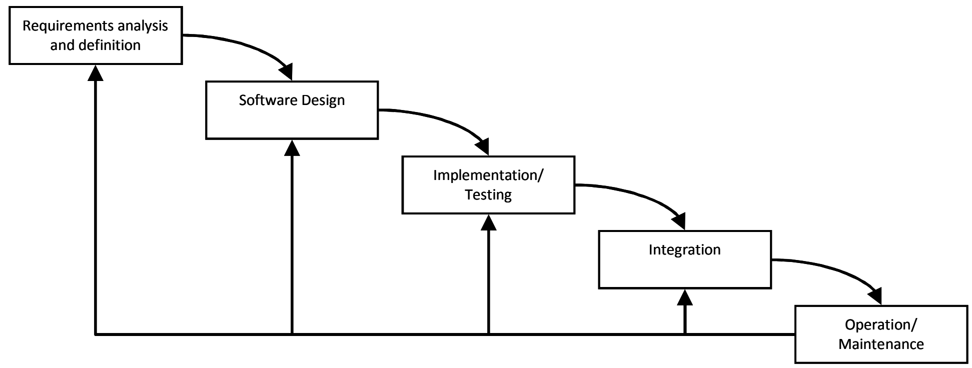


**Figure 1. "Waterfall Model" [Sommerville 2006]**

## A GENERIC SERVICE LIFECYCLE MODEL

Based on these considerations, we defined a more generic service lifecycle model. It mainly consists of seven phases (system analysis, service definition, service design, service development, service deployment, service operation, and service retirement). Excluding service operation, all phases are part of design time – service operation is a runtime phase (cf. Figure 3). We define a feedback loop to allow for stepping back a phase for change purposes, e.g. due to an error that occurred in the preceding phase that must be corrected.

The entry phase is the *system analysis*, where fundamental decisions concerning the system and its scope are made. (Service granularity, e.g., is an aspect that increasingly gains importance when creating services. Many of the above investigated approaches awkwardly neglect an explicit *system analysis* or *requirements analysis* phase.) Once the *service definition* is accomplished, the *design* phase starts, where basic software design procedures concerning, e.g., interface definitions and design principles such as communication standards and code reuse are applied. The following phase is *service development* which basically covers the implementation and intensive service testing.

In service systems, we consider a distinctive *deployment* phase that requires much more attention than in component development or integration in SE. In addition to activities typically performed in the corresponding "integration" phase in the SE cycle, this phase covers final service tests, the installation on a server, which implies the delivery to the service host, registration in service registries or repositories, conformity checks to governance policies, and the resulting final approvals (e.g., description and interface checks).

Once a service is deployed, the service proceeds to the *operation* phase (runtime). This phase basically consists of service monitoring and change management. The latter covers functional change request handling, including service adoption, service versioning, business process change, service portfolio change, etc. In particular, it organizes the proper transition to the service *definition* phase when a change is requested. In this case, the service will be marked as deprecated and a definition process is triggered, i.e., it restarts traversing the lifecycle.

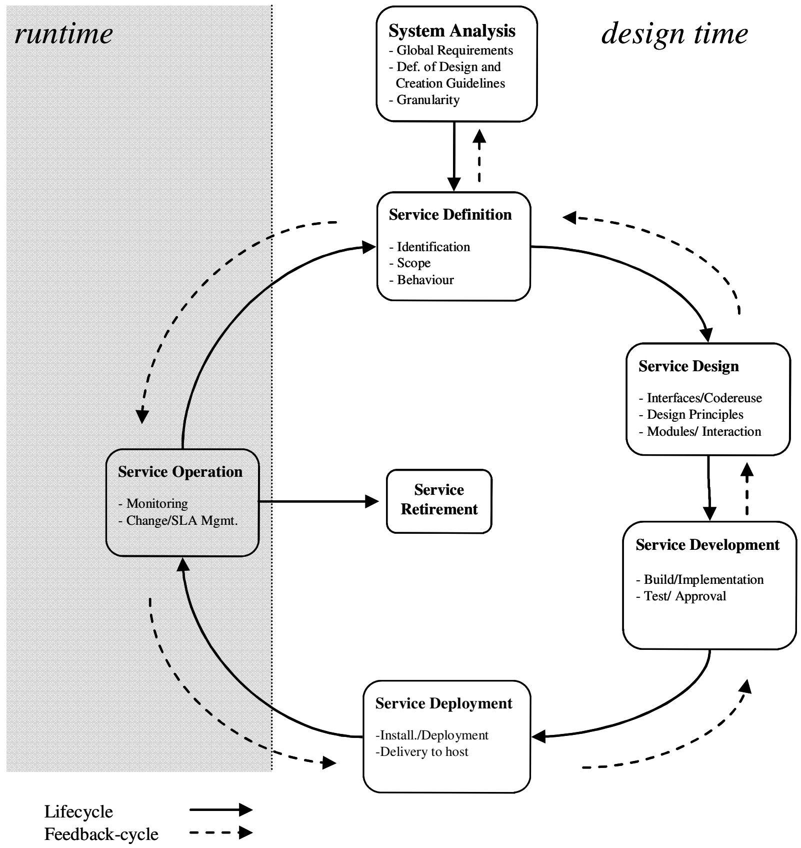


**Figure 3. A Service Lifecycle – Our Approach**

At this point, either a new service *version* is needed – or an additional *service*. In the latter case, a new service lifecycle iteration is started targeting the generation of a new service. In the first case, the service development process is forked. One service instance at its current version continues being operated and is marked as deprecated. A second instance is taken offline and reiterates the lifecycle for improvement. When this instance has passed the service deployment phase, the old deprecated version is gradually replaced by the improved one, i.e., it traverses to the phase *service retirement*. This process of preparing and performing the reiteration of the lifecycle maps to the change time described in the approaches above – however, this process cannot explicitly be described as change phase or time.

This lifecycle model is congruent to the classic SE lifecycle and maps to all investigated approaches. It details most of the shorter cycles such as those by McBride (2007) and Windley (2006). As defined above, it covers the approaches by Woolf (2006), Software AG (2008), Brauer and Kline (2005), and Strnadl (2007), and extends them by the ability to reiterate the cycle. It also summarizes the longer cycles defined by Wall (2006a) and Gu and Lago (2007) by consolidating detailed sub-phases.

As demonstrated, our model maps to the lifecycles that define an explicit change phase or change time by addressing the necessity for change in the service operation phase and the reiteration of the lifecycle. In this respect, all phases in the change time box shown in Figure 2 become part of the phase Operation/Maintenance – the remainder of the change activities are covered by reiterating the service lifecycle.

## CONCLUSION

A central aspect of service governance is the management and definition of a consistent service lifecycle. As shown in this paper, approaches can vary from one extreme to another in structure and definitions. However, almost all approaches contain particularities that reflect experience in addressing the specific needs of a SOA system. Our life cycle approach combines them with the common SE perspective and a discussion that invalidated the need for an explicit change phase or change time. We elaborated a consolidated approach that integrates all hitherto existing findings and allows for the particular requirements of a SOA system. Each of the investigated lifecycle proposals maps to our model.

In spite of the unusual phase names (*assure* meaning *operation*, and *manage* being either design or runtime activity), in the compared lifecycle models, we state that all change phases and change time definitions can completely be replaced by the common definitions of the phase *Operation/Maintenance* combined with lifecycle reiterations. There seems to be no need for an explicit "change time".

## DISCLAIMER

This work is supported in part by the E-Finance Lab e.V., Frankfurt am Main, Germany (www.efinancelab.com).

The project was funded by means of the German Federal Ministry of Economy and Technology under the promotional reference 01MQ07012. The authors take the responsibility for the contents.

## REFERENCES

1. Afshar, M. (2007) SOA Governance: Framework and Best Practices. Oracle Corporation Headquarters, http://www.oracle.com/technologies/soa/docs/oracle-soa-governance-best-practices.pdf, visited January 15, 2009.
2. Bhushan, B. and Tayal, S. (2007) Software Engineering, Laxmi Publications, ISBN: 978-8131804797.
3. Boehm, B.W. (1988). A spiral model of software development and enhancement. IEEE Computer, 21(5), pp.61-72.
4. Boehm, Barry W. (2007) Software Engineering: Lifetime Contributions to Software Development, Management, and Research, Wiley & Sons.
5. Brauer, B. and Kline, S. (2005) SOA Governance: A Key Ingredient of the Adaptive Enterprise, Hewlett Packard, https://h30406.www3.hp.com/campaigns/2005/events/infoworld_soa_forum/soa.php, visited February 16, 2009.
6. Clark, Andrew J. (2005) IT Governance: Determining who decides. EDUCAUSE, Center for Applied Research, Issue 24, http://net.educause.edu/ir/library/pdf/ERB0524.pdf, visited February 20, 2009.
7. Fabini, Martin (2007) Governance fuer komplexe SOA-Unternehmungen In: Starke, Gernot (Ed.); Tilkov, Stefan (Ed.): SOA-Expertenwissen. dpunkt.verlag Heidelberg, pp.309–323.
8. Gu, Q. and Lago, P. (2007) A stakeholder-driven service life cycle model for SOA, in *IW-SOSWE '07: 2nd international workshop on Service oriented software engineering,* New York, USA, pp.1-7.
9. Huhns, M. and Singh, M. (2005) Service-oriented computing: key concepts and principles, IEEE Internet Computing, 9 (1), pp.75–81, http://www.cse.sc.edu/~huhns/journalpapers/V9N1soc.pdf,visited February 16,2009.
10. Jacobson, I. and Bylund, S. (2000) The road to the unified software development process, Cambridge University Press.
11. Kjaer, A. M. (2004) Governance (Key Concepts), Polity Press, ISBN: 0745629792.
12. Krafzig, D., Banke, K., and Slama, D. (2004) Enterprise SOA: Service-Oriented Architecture Best Practices, Prentice Hall.
13. McBride, G. (2007) The role of SOA quality management in SOA service lifecycle management, IBM DeveloperWorks, 2007, http://www.ibm.com/developerworks/rational/library/mar07/mcbride/,visited December 20,2008.
14. Matsumura, M. (2007) The definitive guide to SOA Governance and lifecycle management, WebMethods 2007, http://www.scribd.com/doc/7056281/Guide-to-SOA-Governance,visited February 8,2009.
15. Mukhi, N. K. and Plebani, P. (2004) Supporting policy-driven behaviors in web services: experiences and issues, in *ICSOC '04: Proceedings of the 2nd international conference on Service oriented computing*, pp.322–328, New York, USA.
16. Newcomer E. and Lomow G. (2004) Understanding SOA with Web Services, Addison-Wesley Professional.
17. Niemann, M., Eckert, J., Repp, N. and Steinmetz, R. (2008) Towards a Generic Governance Model for Service-oriented Architectures. In: AIS: Proceedings of the 14th AMCIS, Toronto, Canada.

18. Papazoglou, M. P. (2003) Service-Oriented Computing: Concepts, Characteristics and Directions, in *4th International Conference on Web Information Systems Engineering (WISE 2003)*, pp.3-12.
19. Riedl, C., Böhmann, T., Rosemann, M., Krcmar, H. (2009) Quality management in service ecosystems. In: Information Systems and E-Business Management, Springer Berlin, **7** (2), pp.199-221.
20. Software AG (2007a) CentraSite Governance Edition, http://www.softwareag.com/corporate/images/sag_centrasite_gov_ed_fs_dec07-web_tcm16-33856.pdf,visited February 16, 2009.
21. Software AG (2007b) “Rule your SOA”, http://www.softwareag.com/Corporate/Images/WP%20SOA%20Governance_tcm16-22130.pdf,visited February 2,2009.
22. Software AG (2008): SOA Lifecycle Governance, Software AG, http://documentation.softwareag.com/webmethods/soal8/overview/soa.htm,visited February 19, 2009.
23. Sommerville, I. (2006) Software Engineering, Addison-Wesley Longman, Amsterdam.
24. Strnadl, C. (2007) Bridging architectural boundaries design and implementation of a semantic BPM and SOA Governance Tool, in *ICSOC ’07: Proceedings of the 5th international conference on Service-Oriented Computing*, pages 518–529, Springer, Berlin.
25. Systinet (2006) SOA Governance: Balancing Flexibility and Control within an SOA, http://www.vivit-worldwide.org/uploadedFiles/Product_Centers/Service_Oriented_Architecture_(SOA)/09EX001HPMAY07_Mercury-Systinet_SOA_Governance.pdf,visited February 16, 2009.
26. Wall, Q. (2006a) Understanding the Lifecycle within a SOA: Design Time. Oracle Corporation, http://www.oracle.com/technology/pub/articles/dev2arch/2006/08/soa-service-lifecycle-design.html,visited February 19, 2009.
27. Wall, Q. (2006b) Understanding the Lifecycle within a SOA: Runtime, Oracle Corporation, http://www.oracle.com/technology/pub/articles/dev2arch/2006/11/soa-service-lifecycle-run.html, visited February 19, 2009.
28. Weill, P. and Ross, J. (2004) IT Governance: How Top Performers Manage IT Decision Rights for Superior Results, Harvard Business School Press, June 2004.
29. Windley, P. J. (2006) SOA Governance: Rules of the Game, in: *InfoWorld, v28 n4 p29(5)*, January 2006, http://akamai.infoworld.com/pdf/special_report/2006/04SRsoagov.pdf, visited February 8, 2009.
30. Woolf, B. (2006) Introduction to SOA Governance, International Business Machines Corporation (IBM) 2006, http://www.ibm.com/developerworks/library/ar-servgov/, visited February, 16 2009.