

The Impact of Software Development Strategies on Project and Structural Software Attributes in SOA

Mikhail Perepletchikov, Caspar Ryan, and Zahir Tari

RMIT University
School of Computer Science and Informational Technology
{mikhailp, caspar, zahirt}@cs.rmit.edu.au

Abstract. Service-Oriented Architecture (SOA) is a promising approach for developing integrated enterprise applications. Although the architectural aspects of SOA have been investigated in research and industry literature, the actual process of designing and implementing services in SOA is not well understood. The goal of this paper is to identify tasks needed for successful design and implementation of services, and investigate their effect on the project and structural software attributes in the context of SOA. This facilitates the specification of guidelines for decreasing the required development effort and capital cost of the SOA projects, and improving the structural software attributes of service implementations. The tasks are identified in the context of top-down, bottom-up and meet-in-the-middle software development strategies.

1 Introduction

Service-Oriented Architecture (SOA) is an approach for developing enterprise software systems that employ services. SOA-based systems are defined as a collection of interacting services that offer well-defined interfaces to their potential users, where a service represents a function that is self-contained, and does not depend on the context or state of other services [7].

Although the notion of a “service” is becoming increasingly popular as a means for developing large-scale distributed systems, no systematic, methodological approach to service-oriented software development exists to date [11]. Furthermore, there are conflicting opinions as to which development strategy should be used when developing SOA-based systems. These strategies include top-down, bottom-up, and meet-in-the-middle development approaches, and even though such approaches are applicable to the development of informational systems in general [1], this paper concentrates on additional constraints and properties introduced by SOA.

The contribution of this paper is as follows. Firstly, the general tasks for designing and implementing SOA-based applications were identified based on a critical analysis of related literature [2, 3, 6, 7, 13, 15-17], communication with industry practitioners and researches [16, 17], and the authors’ practical experience with SOA development. Secondly, the impact of these tasks on project and structural software attributes were

analytically determined. Finally, initial guidelines for improving the internal structure of services while decreasing project costs were specified.

The emphasis of this paper is on the design and implementation phases of SOA development rather than enterprise architecture or business modeling. As such, it concerns issues related to the transition from business process models to the implementation of services in software. This lays a foundation for further study of methodological aspects covering design and implementation of SOA-based systems. In addition, the paper briefly discusses the relationship between structural software attributes and software quality attributes. Such relationship will be formalised and evaluated in future work.

The rest of the paper is organised as follows: Section 2 presents background material including important concepts of SOA, descriptions of software and project attributes under investigation, and an overview of top-down, bottom-up, and meet-in-the-middle development strategies. This facilitates identification of development tasks and their impact on project and structural software attributes in the context of SOA, and the provision of guidelines for successful design and implementation of services in SOA as described in Section 3. Finally, Section 4 closes with conclusions and a discussion of future work.

2 Important Characteristics of SOA

SOA is an abstract concept of how software services should be composed and orchestrated. A conceptual model of SOA consists of two primary parties: a service provider, who publishes a service description and realises the service; and a service consumer, who finds the service description in a registry and invokes the service [2].

The notion of a service is similar to that of a component, in that services, much like components, are independent building blocks that collectively represent an application [10]. However, services are coarser grained than components; and they exhibit complete autonomy from other services, meaning that each service is implemented separately from other services resulting in a loosely coupled system [7]. In addition, services can be composed into *composite* services or *business processes*, hence they can be reused in a context not known at the design time.

For the purpose of this paper, SOA is defined as *a software development paradigm that is based on a concept of encapsulating application logic within the independent, loosely coupled, business-aligned services that interact via messages using standard communication protocols*. This particular definition was chosen since it captures the main essence of SOA from both, representational (architectural) and development perspectives.

2.1 Software Engineering Attributes in the Context of SOA

Since the specific software engineering attributes for SOA are yet to be defined, this paper discusses how conventional software engineering attributes can be applied in the

context of SOA-based design and implementation. A software attribute of a product is any feature or property of the product. The attributes used in this paper can be categorised as: project based attributes (including *capital cost* and *development effort*), and software attributes (divided into *internal structural* attributes and *external quality* attributes).

- **Project based attributes**

In traditional SE, the dominant part of the overall project cost is usually the development effort dictated by the estimated size of the final software product [8]. This is not necessarily true for service-oriented development since one of the advantages of SOA is the ability to develop new applications by repurposing pre-existing services, or purchasing services from software vendors. Consequently, development effort might be low when the services are predominantly repurposed or purchased, whereas the actual capital cost can be high depending on the cost of the purchased services.

For the purpose of this paper, *capital cost* is analysed separately from *development effort*, where capital cost represents upfront project costs including: equipment, development tools, and training costs. Development effort represents ongoing costs throughout Software Development Life Cycle.

- **Internal structural software attributes**

The paper investigates the impact of development strategies and their associated activities and tasks on the widely-used [4] internal structural software attributes of coupling, cohesion, and complexity.

In line with its common usage, *coupling* is defined as a measure of the extent to which interdependencies exist between implementation of services in software. *Cohesion* is defined as the extent to which elements of a service contribute to one and only one task. Finally, *complexity* is defined in terms of the internal work performed by a service. In general, low coupling and complexity, and high cohesion are desired [4].

Structural software attributes do not describe visible quality of a product, rather, they have a causal impact on external quality attributes. Identifying guidelines for decreasing complexity and coupling, and increasing cohesion of services ultimately aims to positively influence external quality attributes.

- **External quality attributes**

According to the quality model specified in the ISO/IEC 9126-1:2001 standard [9], there are six main external software quality attributes: functionality, reliability, efficiency, usability, maintainability, and portability.

The structural software attributes combined with various factors influence the external quality attributes, therefore a predictive model for estimating a particular quality attribute can be established in the form of:

$$\text{Quality attribute} = f(\text{structural attributes, other factors})$$

The external quality attributes are introduced in this paper for the purpose of establishing a connection between structural properties of services and quality of SOA-based systems. The derivation of formal, measurable models for each of the external quality attribute will be described in future work.

2.2 SOA Development Strategies

There are three main strategies used for developing SOA-based enterprise applications: top-down, bottom-up, and meet-in-the-middle.

A **top-down** strategy starts with the requirements and business process models and refines them in a stepwise fashion down to a software implementation. The top-down development is often referred to as domain decomposition, which consists of the decomposition of the business domain into its functional areas and subsystems [2]. In the SOA-based top-down development, business process models provide a blueprint for the identification of services. Services are then modeled and realised by service providers, and consumed by service consumers.

A top-down development strategy is arguably more interoperable than a bottom-up approach since avoiding language-specific types and starting with interface and message definitions can lead to a much higher likelihood of interoperability [12]. The drawback of top-down approach is that, in its full generality, it can only be applied to systems developed entirely from scratch [1].

A **bottom-up** strategy, in contrast, originates from the technical basis and tries to work upwards to the requirements and business process models by building services on a top of existing (legacy) systems. In bottom-up development, software engineers analyse and leverage APIs, transactions, and modules from legacy systems such as mainframe or ERP applications. In some cases, componentisation of the legacy systems is needed to re-modularise the existing assets to support service functionality [2]. Most distributed information systems these days involve a component of bottom-up development [1].

A bottom-up strategy includes two different activities. Firstly, developers can *add a layer of services* on top of legacy systems by creating wrappers and adaptors for legacy software. Secondly, legacy systems can be *refactored* in such a way that the external behavior of the code remains the same, whereas the internal structure becomes SO.

A **meet-in-the-middle** strategy is essentially a combination of top-down and bottom-up techniques. Currently, the techniques for meet-in-the-middle approach are not well understood. To the knowledge of the authors, the only well-described technique is a goal-service modeling proposed by Arsanjani [2].

In this technique, high-level business process functionality is externalised for coarse-grained services. Examining the existing legacy functionality and deciding how to create adaptors and wrappers allows specifying finer-grained services. Finally, a cross-sectional approach can be applied in order to reduce the number of candidate services that have already been identified. This technique also ties services to goals, performance indicators, and metrics.

3 The Impact of Development Strategies on Software Attributes

In order to facilitate investigation of the impact of development strategies on project and structural software attributes, the top-down, bottom-up, and meet-in-the-middle strategies have been divided into a number of general *activities*, where an activity contains a number of *tasks* for designing and implementing services in SOA-based

applications. Tables 1-3 show the development strategies together with the associated activities and tasks, where a grouping of related tasks is shown in a separate cell within the table.

The impact of the identified tasks on the project and structural software attributes was analysed, and tasks have been grouped together based on their influence on a particular attribute under investigation. *The up (↑) and down (↓) arrows are used to indicate the impact of a particular task on the attributes under investigation. The 🍷 and 🍷 symbols are used to indicate whether such impact is positive or negative in regards to a particular attribute.*

In situations where a task influences attribute/s other than the one it was originally intended for, the impact of this task on such attribute/s is shown in brackets together with arrows indicating negative/positive influence. For example, the ‘provide training’ task in the “Building services” activity of the top-down strategy directly influences the *capital cost* attribute thus having following indicator associated with it - ↑ (**Development Effort** ↓). *This states that the ‘provide training’ task will increase capital cost, but at the same time decrease development effort.*

Any given combination of tasks constitutes a *guideline* that can be selected based on the requirements of the project. Note that the aim of this paper is not to identify a concrete development methodology, but to investigate the impact of tasks on project and software attributes, and establish initial guidelines for SOA-based design and implementation. These guidelines are presented in the following sub-sections.

3.1 Guidelines for Top-down Development

There are various activities involved in realising services in a top-down approach. Such activities include *building* services from scratch, *repurposing* existing services, and *purchasing* services.

The crucial task of building services from scratch is to identify the smallest units of software (service components) that can be reused in different contexts. Service components should be then composed into coarser-grained composite services or business processes. By structuring the system as a set of highly-reusable, loosely-coupled services, companies can increase Return on Investment (ROI) due to decreased maintenance costs and ability to repurpose services in future projects.

Also, organisations should purchase Enterprise Service Bus (ESB) implementations to facilitate connectivity, routing of messages, etc. In addition, Integrated Services Environments (ISE) should be used to design, configure, test, and debug business processes. Although these products might increase the capital cost of the project, they will reduce the required development effort as shown in Table 1.

To facilitate the future repurposing of services, an enterprise should incorporate a private service registry to centralise published service descriptions into one accessible resource. When repurposing services, pre-existing services should be integrated into the system using integration/composition code, the services themselves should not be modified. This will save time on testing since there is no need to conduct unit tests on the pre-existing services, only integration tests are required. Finally, prior to making a decision to purchase services, an enterprise should conduct a Cost-Benefit Analysis to evaluate pros and cons of purchasing services instead of building them in-house.

Table 1. The impact of *top-down* strategy on the project and structural software attributes

Attributes Activities	Project		Structural Software		
	Capital Cost ↓👍 (CC) ↑👎	Devel. Effort ↓👍 (DE) ↑👎	Complexity ↓👍 (C1) ↑👎	Coupling ↓👍 (C2) ↑👎	Cohesion ↑👍 (C3) ↓👎
Building services	<ul style="list-style-type: none"> - Have existing team of developers ↓ - Provide training ↑ (DE ↓) - Purchase standardized middleware and development tools (eg. ESB) ↑ (DE ↓) - Establish standard documentation/reference models ↑ (DE ↓) - Maintain private registry of services ↑ (ROI ↑) 	<ul style="list-style-type: none"> - Build iteratively ↓ - Use mature software development processes ↓ (CC ↑) - Group development teams around logical business tasks ↓ - Build for reuse ↑ (ROI ↑ C2 ↓) 	<ul style="list-style-type: none"> - Apply MDA approach to decompose business processes (BP) into fine-grained service components ↓ (DE ↓) - Implement service components using principles of OO ↓ - Decompose highly-complex components ↓ - Encapsulate global data in a dedicated service ↓ 	<ul style="list-style-type: none"> - Identify the smallest units of software that can be reused in different contexts (service components) ↓ - Couple service components and services through interfaces only, not through implementation ↓ - Specify simple, concise interfaces ↓ - Avoid embedding workflow aspects within services implementation ↓ 	<ul style="list-style-type: none"> - Develop fine-grained service components ↑ - Compose service components into composite services only if resulting service represents a concrete business function ↑ - Avoid embedding application policies such as security, SLAs, and QoS within services themselves ↑
Repurposing services	<ul style="list-style-type: none"> - Hire a business modeling expert to identify existing services that can be reused in new application ↑ (DE ↓) - Utilise existing middleware and development tools ↓ 	<ul style="list-style-type: none"> - Reuse preexisting services identified from private registry ↓ - Embed composition code necessary to support new capabilities into BPs, not in individual services ↓ 	N/A (the internal structure of services remains intact)	N/A (the internal structure of services remains intact)	N/A (the internal structure of services remains intact)
Purchasing services	<ul style="list-style-type: none"> - Purchase services from known vendors ↑ (DE ↓) - Perform adequate Cost-Benefit Analysis to evaluate the costs and benefits of developing /purchasing services ↑(DE ↓) 	<ul style="list-style-type: none"> - Develop only specific services, purchase the rest ↓ - Purchase fine-grained services, but build coarse services in-house ↓ - Repurpose if possible ↓ 	N/A (we cannot influence the internal structure of purchased services)	N/A (we cannot influence the internal structure of purchased services)	N/A (we cannot influence the internal structure of purchased services)

3.2 Guidelines for Bottom-up Development

An important task in bottom-up development is to use software quality metrics to measure the structural design properties of legacy systems in order to decide whether it is best to *refactor the system*, or simply *add a layer of services* to it. In future work, the suitability of existing structural complexity measures will be evaluated, and a threshold for acceptable level of complexity will be established. Also, it is important to take business process models into account when determining required services.

When refactoring legacy systems, it is advisable to start small, focusing on strongly-coupled and highly-complex modules. This will allow measuring ROI before making a large commitment, and gain experience before taking on larger problems. To reduce development cost when refactoring existing systems, an organization should make an effort to employ people who were involved in the architecture, design, and implementation of such systems as shown in Table 2. To reduce development cost when adding a layer of services to legacy systems, companies should consider purchasing commercial off-the-shelf software service adaptors/wrappers. In addition, the existing resources should be utilised as much as possible.

The main factor influencing the internal structural properties of services in bottom-up development is the granularity of services. Developers should make an effort to develop fine-grained services, consequently increasing cohesion, and decreasing complexity and coupling.

3.3 Guidelines for Meet-in-the-middle Development

The bottom-up approach can lead to poor business-service abstractions since the design is usually dictated by the existing IT environment, rather than business needs. On the other hand, a top-down strategy might cause insufficient, non-functional requirement characteristics, and provide an impedance mismatch on the service and component layer [17]. Therefore, a meet-in-the-middle strategy is highly recommended.

The meet-in-the-middle is potentially the most expensive approach, but should result in a more-complete set of business-aligned services, consequently increasing ROI as shown in Table 3. The tasks for improving structural software properties in a meet-in-the-middle development include a combination of previously-described guidelines for top-down and bottom-up software development strategies.

3.4 Conflicting Factors

There are a number of conflicting factors that negatively influence some of the attributes, while contributing positively to others. Such factors introduce trade-offs between project cost and software quality, hence they should be carefully analysed by managers and software engineers in order to decide on a particular course of action.

Table 2. The impact of *bottom-up* strategy on the project and structural software attributes

Attributes Activities	Project		Structural Software		
	Capital Cost ↓👍 (CC) ↑👎	Devel. Effort ↓👍 (DE) ↑👎	Complexity ↓👍 (C1) ↑👎	Coupling ↓👍 (C2) ↑👎	Cohesion ↑👍 (C3) ↓👎
Refactoring legacy systems	<ul style="list-style-type: none"> - Employ people who were involved in the architecture /design of legacy systems ↑ (DE ↓) - Purchase utility (general-purpose) services ↑ (DE ↓) - Maximise use of existing resources (eg. DBs) ↓ 	<ul style="list-style-type: none"> - Refactor iteratively ↓ - Focus on strongly-coupled and highly complex modules ↓ (C1 ↓ C2 ↓ C3 ↑) - Purchase service adapters for modules that are loosely-coupled and highly cohesive (no refactoring needed) ↓ (CC ↑) 	<ul style="list-style-type: none"> - Share complexity across refactored service components ↓ 	<ul style="list-style-type: none"> - Remove implementation coupling by ensuring that refactored modules and modules with service adaptors communicate strictly through the interfaces ↓ 	<ul style="list-style-type: none"> - Refactor existing modules into fine-grained service components ↑
Adding a layer of services to legacy systems	<ul style="list-style-type: none"> - Employ people who were involved in the architecture /design of legacy systems ↑ (DE ↓) - Use COTS service adaptors ↑ (DE ↓) - Maximise use of existing resources ↓ 	<ul style="list-style-type: none"> - Develop coarse-grained services ↓ (C1 ↑ C2 ↑ C3 ↓) - Establish ESB and incrementally add services to it ↓ - Remove dependencies between systems that share infrastructure ↑ (C1 ↓ C2 ↓) 	<ul style="list-style-type: none"> - Legacy systems should interact only through service layer ↓ 	<ul style="list-style-type: none"> - Avoid combining functionality from different legacy systems into one service ↓ 	<ul style="list-style-type: none"> - Add fine-grained services ↑

Table 3. The impact of *meet-in-the-middle* strategy on the software attributes

Attributes Activities	Project		Structural Software		
	Capital Cost ↓👍 (CC) ↑👎	Devel. Effort ↓👍 (DE) ↑👎	Complexity ↓👍 (C1) ↑👎	Coupling ↓👍 (C2) ↑👎	Cohesion ↑👍 (C3) ↓👎
Adding a layer of services to legacy systems	<ul style="list-style-type: none"> - Employ people who were involved in the architecture /design of legacy systems ↑ (DE ↓) - Establish standard documentation models ↑ (DE ↓) - Maximise use of existing resources ↓ 	<ul style="list-style-type: none"> - Examine legacy systems to determine services that can be developed by externalising existing functionality ↓ - Apply cross-sectional approach [2] to cut down the number of candidate services ↓ 	<ul style="list-style-type: none"> - Combination of top-down and bottom-up approaches 	<ul style="list-style-type: none"> - Combination of top-down and bottom-up approaches 	<ul style="list-style-type: none"> - Combination of top-down and bottom-up approaches

Two major conflicting factors were identified: Firstly, the *build for reuse* task in the “Building services” activity of the top-down strategy results in higher development effort, but at the same increases ROI and improves implementation-level coupling of services as shown in Table 1. Hence, a trade-off between increased reusability and higher development cost can be observed. This is due to the fact that building a reusable unit (service) requires three to five times the effort needed to develop a unit (service) for one specific purpose [5]. On the other hand, highly-reusable services can decrease future development costs, consequently increasing ROI. Also, highly-reusable services will exhibit low coupling since they are built as totally independent software units. When building for reuse, project managers should consider these issues, so that an informed decision can be made regarding development for reuse.

Secondly, the granularity of services influences a number of attributes. For example, *developing coarse-grained services* when adding a layer of services to legacy systems will decrease the development efforts since it is easier for developers to generalise existing functionality into coarse-grained service interfaces. Also, coarse-grained services can improve network performance since they require less communication than fine-grained services. On the other hand, creating coarse-grained services introduces increased coupling and decreased cohesion [14], resulting in lower system quality in terms of maintainability, reliability, and efficiency. Therefore, project managers should make a trade-off in regards to expected granularity of services based on the particular project constraints.

4 Conclusions and Future Work

This paper has identified general tasks for the design and implementation phases of SOA-based development in the context of top-down, bottom-up, and meet-in-the-middle strategies. The impact of such tasks on project and structural software attributes has been qualitatively analysed. The tasks were combined into general *guidelines* for improving the internal structure of SOA-based software, and decreasing capital cost and development effort. Although the guidelines presented in this paper have not been empirically evaluated, they could be used by project managers and software engineers in order to determine a suitable development approach given particular quality requirements, project constraints, and application types.

To formalise findings presented in this paper, a suite of SOA-oriented metrics for measuring and quantifying project and software quality attributes will be identified in future work. Such metrics will be applied to the data collected from available SOA-based projects, consequently facilitating an empirical evaluation of the presented guidelines.

In addition, the issues discussed in the paper should facilitate future research into design and implementation of services in SOA. For example, the paper described two of the main issues related to SOA-based development that need to be investigated in future work: i) can services be made sufficiently independent so as to be reused in entirely different applications, whilst minimising development effort?; and ii) what is the optimal granularity of services?

Finally, the recommendations for *directly* influencing external quality attributes during the development will be provided in future work. For example, to increase *efficiency*, an organisation could develop/purchase service-oriented messaging backbone to communicate in formats other than XML since XML parsing and manipulation are very resource consuming.

Acknowledgement: This project is funded by the ARC (Australian Research Council), under Linkage scheme no. LP0455234.

References

- [1] Alonso, G., et al., Web Services: Concepts, Architectures and Applications. 2004, Heidelberg, Germany: Springer-Verlag.
- [2] Arsanjani, A., Service-oriented modeling and architecture: how to identify, specify, and realize services for your SOA. 2004, IBM - whitepaper.
<ftp://www6.software.ibm.com/software/developer/library/ws-soa-design1.pdf>
- [3] Barry, D.K., Web services and service-oriented architectures: the savvy manager's guide. 2003, San Francisco, CA: Morgan Kaufmann; Elsevier Science.
- [4] Briand, L.C. and J. Wust, Modeling development effort in object-oriented systems using design properties. IEEE Transactions on Software Engineering, 2001. 27(11): p. 963-986.
- [5] Crnkovic, I. Component-based Software Engineering. in 25th International Conference on Information Technology Interfaces. 2003. Cavtat, Croatia.
- [6] Endrei, M., et al., Patterns: Service-Oriented Architecture and Web Services. 2004: IBM Redbooks.
- [7] Erl, T., Service-Oriented Architecture: a field guide to integrating XML and Web services. 2004, Upper Saddle River, NJ: Prentice Hall PTR.
- [8] Fenton, N.E. and M. Neil, Software Metrics: Roadmap, in Future of Software Engineering, A. Finkelstein, Editor. 2000, ACM Press.
- [9] ISO/IEC, 9126-1:2001 Software Engineering: Product quality - Quality model. 2001.
- [10] Kotonya, G., et al. A service model for component-based development. in 30th EUROMICRO Conference. 2004. Rennes, France.
- [11] Kruger, I.H. and R. Mathew. Systematic development and exploration of service-oriented software architectures. in Fourth Working IEEE/IFIP Conference on Software Architecture. 2004. Oslo, Norway.
- [12] Lehmann, M., Deploying large-scale interoperable Web Services infrastructures, in Web Services Journal. 2005. p. 10-15.
- [13] Papazoglou, M.P. Service-Oriented Computing: concepts, characteristics and directions. in International Conference on Web Information Systems Engineering. 2003. Roma, Italy.
- [14] Perepletchikov, M., C. Ryan, and K. Frampton. Comparing the Impact of Service-Oriented and Object-Oriented Paradigms on the Structural Properties of Software. in Second International Workshop on Modeling Inter-Organizational Systems (MIOS'05). 2005. Ayia Napa, Cyprus.
- [15] Singh, M.P. and M.N. Huhns, Service-Oriented Computing: Semantics, Processes, Agents. 2005, West Sussex, England: John Wiley & Sons.
- [16] Yang, J., M.P. Papazoglou, and B. Orriens, Service component: a mechanism for Web Service composition reuse and specialization. Journal of Integrated Design and Process Science, 2004. 7(4): p. 1-18.
- [17] Zimmermann, O., P. Krogh, and C. Gee, Elements of Service-Oriented Analysis and Design: an interdisciplinary modeling approach for SOA projects. 2004, IBM - whitepaper.
<http://www-128.ibm.com/developerworks/library/ws-soad1/>