Vysoká škola ekonomická v Praze

Fakulta informatiky a statistiky

Katedra informačních technologií

Student	:	**Daniel Hejl**
Vedoucí bakalářské práce	:	**doc. Jiří Feuerlicht, Ph.D.**
Oponent bakalářské práce	:	**Ing. Jarmila Pavlíčková**

TÉMA BAKALÁŘSKÉ PRÁCE

Evaluating Quality of Service Design

ROK : 2011

Prohlášení

Prohlašuji, že jsem bakalářskou práci zpracoval samostatně a že jsem uvedl všechny použité prameny a literaturu, ze kterých jsem čerpal.

V Praze dne 29.06.2011 ....

podpis

Acknowledgements

I would like to thank to my thesis supervisor doc. Jiří Feuerlicht, Ph.D. for all advices that helped me to stay on the track and complete this thesis in time.

Also, I would like to thank to my employer, company JetMinds s.r.o., for understanding for my study obligations and for providing excellent working environment.

Abstract

Service Oriented Architecture is a popular choice of system architecture and many organization are moving towards it in order to closely align IT capabilities with their business goals. They are expecting that the implementation of SOA will lead to increased efficiency, the ability to rapidly respond to changing business environments, and significantly improved return on investment. In order to meet these expectations, great emphasis must be given on correct service design. However, designing services in the way that they are highly reusable and the final product is highly maintainable can be especially challenging. This bachelor thesis focuses on summarization of how metrics for measuring structural properties of service oriented software designs can be used for predicting potential problems in service design. In the practical part of this thesis, a tool for evaluating a service design based on coupling between service interfaces was implemented and practical applicability of one of the metric was demonstrated.

Abstrakt

Architektura orintovaná na služby (SOA) je populární volbou systémové architektury a mnoho organizací na ni přechází aby lépe sladily možnosti IT s jejich podnikovými cíly. Mezi očekávání, která adopce SOA vyvolává patří například zvýšená efektivita, schopnost rychle reagovat na změny v obchodním prostředí a zlepšení návratnosti investic. Aby tato očekávání byla naplněna, velký důraz musí být kladen na správný návrh služeb. Navrhování služeb takovým způsobem, aby měly velkou míru znovupoužitelnosti a aby výsledný produkt byl dobře udržovatelný může být ale obzvláště náročné. Tato práce shrnuje způsoby, jakými se dají využít metriky měřící vlastnosti struktur návrhu systémů orientovaných na služby pro předpovídání problémů v návrhu služeb. V praktické části této bakalářské práce byl vyvynut nástroj na měření kvality návrhu služeb na základě jedné z metrik a byla demonstrována praktická použitelnost této metriky.

Table of Contents

Illustration Index

Index of Tables

1 Introduction

Enterprise information systems are becoming increasingly large and complex requiring more precise mechanisms for managing software complexity and, more importantly, meeting the demands of highly-dynamic business environments. In order to efficiently support these objectives, the Service Oriented Computing (SOC) paradigm was introduced as an extension to the existing development approaches (such as Procedural and OO development)[16].

SOC provides a flexible and agile development model by introducing an additional layer of software abstraction – a service layer. Service-oriented applications are structured as a collection of independent, business-aligned software services, which can be composed into executable business processes. The business processes encapsulate business logic and rules, separating them from the software implementation of services, thus promoting higher reusability of the individual services and facilitating rapid propagation of business changes and reduction of maintenance efforts [16].

Service Oriented Architecture (SOA) is a popular choice of system architecture and many organization are moving towards it in order to closely align IT capabilities with their business goals. They are expecting that the implementation of SOA will lead to increased efficiency, the ability to rapidly respond to changing business environments, and significantly improved return on investment.

The shift towards SOA is usually connected with adopting technologies based around Web Services core standards (e.g. XML, SOAP, WSDL, and UDDI, and the various WS-.* extensions). The universal acceptance of these technical standards by the industry produced a situation where for the first time it is technically possible for applications to interact across diverse computing environments without incurring massive integration costs. However, implementing the system using these standards alone does not ensure that the business benefits of SOA will be fully realized [1].

The adoption of technology standards is a necessary prerequisite for achieving low cost integration between disparate technology platforms, but other issues that include service analysis and design and agreement on data structures and semantics are equally important.

Service design has been the subject of intense research interest and there is a wide agreement about the key principles that lead to good quality design of services. However, there is evidence that achieving good quality design of services in practice is difficult and that many service oriented applications suffer from low levels of reuse and are difficult to evolve [3].

It is essential to identify the potential problems in service design as early in the Software Development Lifecycle as possible so the necessary improvements can be made.

In this thesis I will investigate how the structural properties of service oriented design can be used for predicting quality characteristics (such as maintainability) of the final service oriented product and how we can effectively quantifies these properties.

1.1 What are the goals of this thesis?

This thesis consists of two distinct parts. The goal of the first part (chapters 2 to 5) is to identify quality characteristic of service design, especially these that could have direct relationship to the reasons adopting SOA solution could fail to meet the expectations as discussed in the first chapter, and investigate how structural properties of service oriented design artifacts can be used for predicting these quality characteristics of the final service oriented product and how we can effectively quantify these properties, especially by reviewing available metrics for measuring these quality characteristics.

During the review, the metrics will be categorized based on the structural property they are quantifying, and several observations about them will be made. For each of the metrics, following will be summarized:

i) Relation of the metric to category/type of structural property it quantifies

ii) Relation of the metric to quality characteristics

iii) Method of measurement

Discussion about applicability of the metric in practice will be done after each set of metrics as these metrics are usually expected to be used together and there is no meaning in discussing them separately. However, direct comparison of precision of these metrics was not made as this would require broader range of industrial scale experiments and analyses.

The metrics were chosen after careful literature review. For searching relevant resources, Google Scholar and ACM Digital Library were used. The requirements for choosing metrics were following:

i) They were created specially for Service-Oriented environment

ii) They are measuring at least one of the structural properties discussed in chapter 4

The goal of the second part of this bachelor thesis (chapter 6) is to implement a tool for evaluating a service design metrics based on coupling between service interfaces. Implementing a tool for other metrics and testing them in practice is out of scope of this thesis.

1.2 Contribution

The contribution of this bachelor thesis is the summarization of current state-of-the-art knowledge about quantifying structural properties of service design in order to predict quality characteristic of the final product and to detect possible problems in early phases of development process.

The contribution of practical part of this thesis is in implementing a tool for evaluating a service design metrics based on coupling between service interfaces and evaluating the DCI metric based on practical test.

1.3 Structure of the thesis

The remainder of this thesis is separated into five chapters.

Chapter 2 covers the basic characteristics of Service-Oriented Architecture and Service-Oriented Computing. The expectations that are connected with adoption of SOA.

Chapter 3 covers quality characteristics of service oriented software products and discuss the relation between those quality characteristics and meeting the expectations discussed in chapter 2.

Chapter 4 covers structural properties of Service Oriented Software Designs and analyses relation between those structural properties and quality characteristics of final service oriented software products.

Chapter 5 reviews available metrics for measuring structural properties of service oriented software designs and discuss the practical applicability of those metrics for improving the quality characteristics of final service oriented software products.

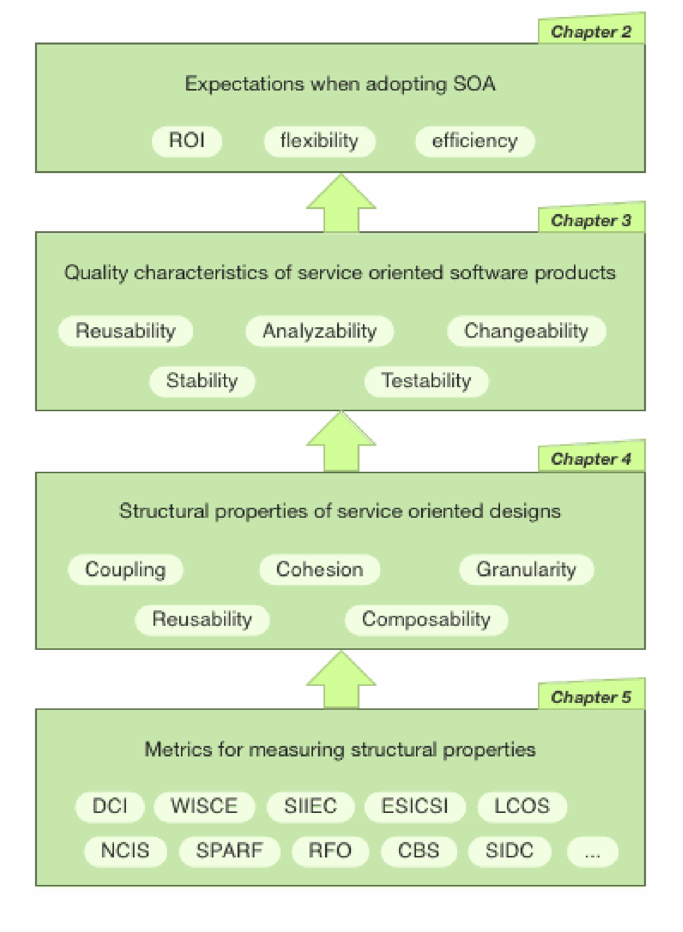

Illustration 1: Structure of the theoretical part of the thesis, source: author

Chapter 6 represents the practical part of this thesis. In this chapter a tool for automatic computing of DCI metric is presented and used for empirical evaluation of the metric on real-life data.

Finally, Chapter 7 presents concluding remarks and improvement suggestions.

2 Service-Oriented Architecture and Service-Oriented Computing

In this section basic characteristics of Service-Oriented Architecture (SOA) and Service-Oriented Computing (SOC) will be very briefly discussed.

2.1 Service-Oriented Computing

According to [12], Service-oriented computing is an umbrella term used to represents a new generation distributed computing platform. As such, it encompasses many things, including its own design paradigm and design principles, design pattern catalogs, pattern languages, a distinct architectural model, and related concepts, technologies, and frameworks.

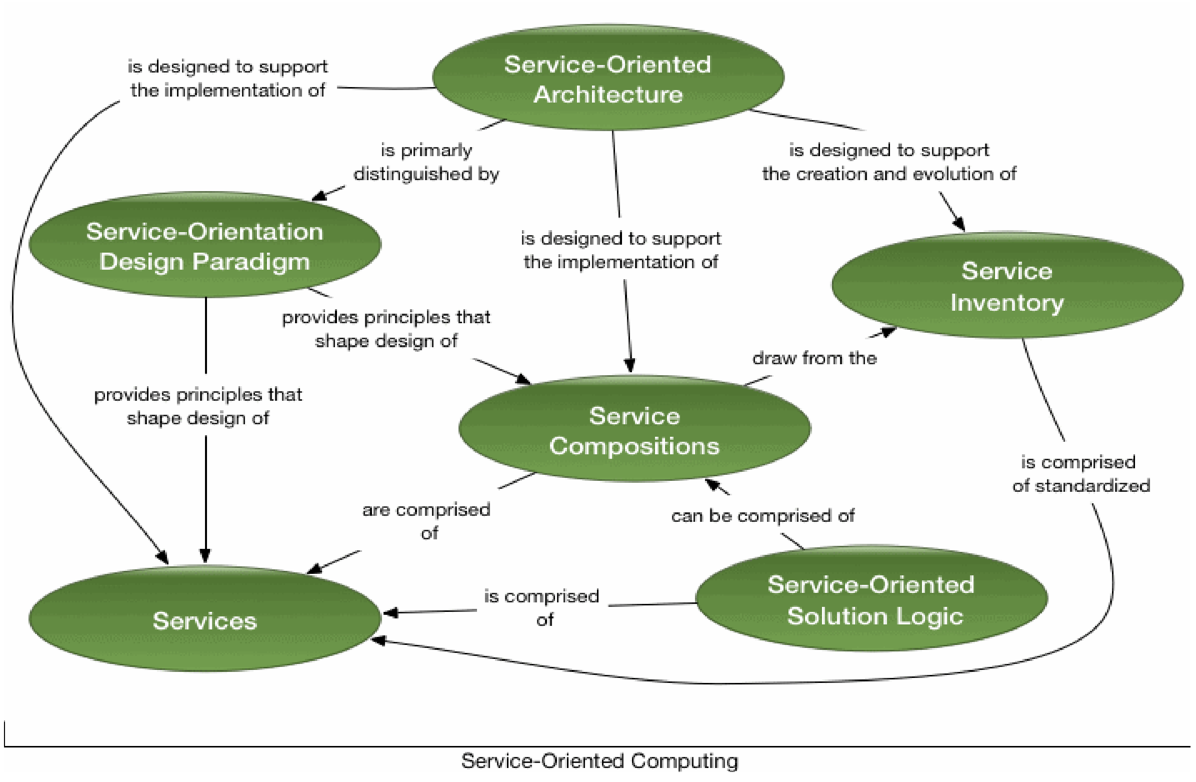
is designed to support the implementation of
Service-Oriented Architecture
is primarly distinguished by
is designed to support the creation and evolution of
Service-Orientation Design Paradigm
is designed to support the implementation of
Service Inventory
provides principles that shape design of
draw from the
provides principles that shape design of
Service Compositions
is comprised of standardized
are comprised of
can be comprised of
Service-Oriented Solution Logic
Services
is comprised of
Service-Oriented Computing

Illustration 2: The elements of Service-Oriented Computing; inspired by [12]

Service-oriented computing builds upon past distributed computing platforms and adds new design layers, governance considerations, and a vast set of preferred implementation technologies.

2.2 Service-Oriented Architecture

Service-Oriented Architecture (SOA) is a technology architectural model for service-oriented solutions with distinct characteristics in support of realizing service-orientation and the strategic goals associated with service-oriented computing [14].

SOA is build around the concept of a service as a basic building block of the distributed systems. These services are collections of capabilities. They consists of logic designed to carry out these capabilities and a service contract that expresses which of the capabilities are made available for public invocation.

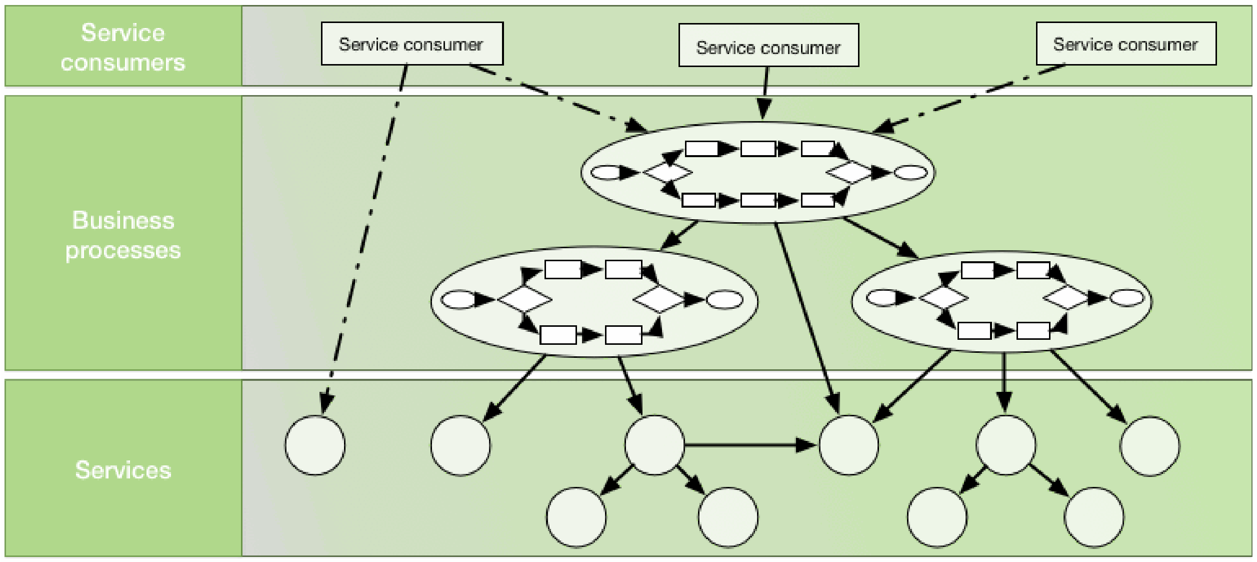

Illustration 3: Architectural view on SOA, source: author

These services can be invoked directly by service consumer or from business processes. Business processes reflect workflows within and between organizations. SOA also provides a way for consumers of services to be aware of available services.

Service Oriented Architecture emphasizes following principles [13]:

- Standardized Service Contract – Services adhere to a communications agreement, as defined collectively by one or more service-description documents.
- Service Loose Coupling – Services maintain a relationship that minimizes dependencies and only requires that they maintain an awareness of each other.
- Service Abstraction – Beyond descriptions in the service contract, services hide logic

from the outside world.

- Service Reusability – Logic is divided into services with the intention of promoting reuse.
- Service Autonomy – Services have control over the logic they encapsulate.
- Service Granularity – A design consideration to provide optimal scope and right granular level of the business functionality in a service operation.
- Service Statelessness – Services minimize resource consumption by deferring the management of state information when necessary
- Service Discoverability – Services are supplemented with communicative meta data by which they can be effectively discovered and interpreted.
- Service Composability – Services are effective composition participants, regardless of the size and complexity of the composition.

SOA benefits include reduction of integration costs, improved business agility and flexibility, improved asset reuse, and most importantly improved return on investment [1].

2.3 Benefits of SOA adoption

According to analysts, SOA benefits include reduction of integration costs, improved business agility and flexibility, improved asset reuse, and most importantly improved return on investment.

The return on investment is affected mainly by improved agility of the organization, lower implementation costs, shortened development and testing periods, faster custom application development, quicker and more cost-effective response to changing market conditions.

2.4 SOA design elements

In order to understand further discussed structural properties of service oriented design and metrics for quantifying those properties, it is essential first to understand elements of Service-Orientated Architecture.

Services are cohesive and autonomous units of business logic. Each service consist of two distinct parts: *service interfaces* and service *implementation elements*.

Each method of the service implementation elements that is exposed in a service interface is considered to be a *service operation*.

An operation has a set of *request and response messages* that are used as data containers between the service consumers and the service.

Request and response messages are aggregations of *schema elements* that constitute the underlying data model.

Service-oriented *process* consists of collection of services. The process itself is represented by series of process activities that are coordinated through a body of workflow logic that is expressed within a process service.

3 Quality characteristics of Service Design

Adopting SOA solution brings a lot of expectations. However, these expectations are not usually fully realized in the real life. There could be many reasons for this, but some of them could be found in Service Design. Designing services in the way that they are highly reusable and the final product is highly maintainable is especially challenging. Let's look on those quality characteristic more closely, as we will be referring to them in the rest of the thesis.

3.1 Reusability

Reusability is the ability of given software module to be used in different situations to add new functionalities with slight or no modifications. Service reusability can be defined as the ability to participate in multiple service assemblies (compositions) [1].

Reuse is regarded by many organizations as the top driver for the adoption of SOA. Service reusability is essential for organizations to achieve good return on investment [16]. From a business point of view there is a direct relationship between reuse and ROI (Return on Investment); reuse reduces the costs associated with design, development and testing, as well as significantly reduces maintenance effort [12].

However the mechanism for achieving reusability of services is poorly understood at present and there is evidence that design of services for reuse is not a prime objectives when implementing SOA [1]. The design of services is driven primarily by performance and scalability considerations, rather than any sound software engineering principles. However so far the perception of improved reuse can be mainly attributed to the ability to derive business value from legacy applications by externalizing existing functionality as Web Services. While reusing functionally locked in legacy applications is clearly important, it is the reusability of services in newly developed applications that will ultimately determine the long-term business benefits derived from SOA [1].

3.2 Maintainability

Software maintainability is one of the most important quality characteristics, representing the capability of the software product to be modified. Modifications can include corrections, improvements or adaptations of the software to changes in environment, and in functional specifications.

The time needed to complete software maintenance activities can play a major role when determining the capability of enterprises to adjust to changing market conditions and to implement innovative products and services in order to stay competitive. This is especially crucial for an emerging generation of constantly-evolving service-oriented enterprise applications.

Although the reported numbers vary, it has been estimated by various researchers that the maintenance phase of the SDLC consumes more than 60% of the overall project resources [16]. Therefore, developing software that is difficult to maintain can contribute to project failures due to the cost and time overruns.

Software maintainability can be subdivided into four sub-characteristics: analyzability, changeability, stability and testability [17].

3.2.1 Analyzability

Analyzability is the capability of the software product to be diagnosed for deficiencies or causes of failures in the software, or for the parts to be modified to be identified. Analyzability is directly related to the amount of time developers need to spend with analyzing the software.

3.2.2 Changeability

Changeability is the capability of the software product to enable a specified modifications to be implemented. Changeability is directly related to the amount of time developers need to spend with coding, designing and documenting changes.

3.2.3 Stability

Stability as a sub-characteristic of maintainability is the capability of the software product to avoid unexpected effects from modifications of the software. Stability is directly related to the amount of time developers need to spend with modifying the parts of the software they initially didn't want to change (e.g. in order to keep the old functionality of those parts).

3.2.4 Testability

Testability is the capability of the software to be validated. Testability is directly related to the effort needed to made in order to validate the software.

3.3 Overview

In this section, we have summarized important quality characteristic of service design - reusability, analyzability, changeability, stability and testability.

We have showed that there is a direct relation between these quality characteristic and ROI as they directly influence the amount of effort needed to be delivered.

In chapter 4, we will analyze more closely the relationship between these quality characteristics and structural properties of service oriented software design (like service granularity or coupling).

4 Structural Properties of Service Oriented Software Designs

The design of any software product possesses a number of properties that can be assessed by measuring the structure of the design artifacts using software metrics. Such structural properties are said to capture the (internal) quality of software and are commonly referred to as internal quality characteristics since they do not describe the visible quality of a product, rather, they have a causal impact on the (external) quality characteristics such as maintainability, reliability, and performance [16].

Five major structural properties that are emphasized by SOA design principles are coupling, cohesion, granularity, reusability and composability [4].

4.1 Coupling

4.1.1 Definition

In computer science, coupling is the degree to which each program module relies on each one of the other modules [12]. It describes the strength of a connection or the nature of the dependencies that exist between one module and another.

Loosely coupled systems are easier to maintain, since a change in one system entity will have less impact on other entities. They are also easier to comprehend (*analyzability* is increased), reuse (*reusability* is increased) and test (*testability* is increased). Low coupling is thus fundamental to the design of any software system, including those that are service-oriented.

On the other side, tight coupling means that related system entities have to know internal details of each other, and have to reflect the changes of the other entities in order to be able to cooperate with them, which reduces *stability* of the system.

4.1.2 Coupling relationships

Since services are composite structures, various types of coupling relationships can occur. According to [16], these relationships can be divided into following categories:

4.1.2.1 Intra-service relationships

Intra-service coupling occurs between implementation elements belonging to the same service. This type of coupling can be considered as the generic type of design coupling and can be linked to the notion of coupling in Procedural and OO designs. This is because an individual service can be considered as a Procedural or OO sub-system when investigated in isolation from the other services in the system; therefore, the impact of intra-service coupling on maintainability is expected to be similar to that suggested for the Procedural/OO systems. High intra-service coupling will result in decreased analyzability, changeability, and stability of a service.

4.1.2.2 Indirect extra-service relationships

Indirect extra-service coupling covers the relationships between services in the system through service interfaces only.

This type of coupling can be considered as the desirable form of (loose) coupling, because services in Service-Oriented Computing should communicate with one another via interfaces in order to achieve some desired functionality. This type of coupling is unavoidable in practice.

If service s_1 has an indirect extra-service relationship to service s_2 (one of the implementation elements of the service s_1 are calling the service s_2 via its interface), the stability and stability of the service s_1 will be decreased, since there is a possibility that changes to the operations exposed in interfaces of the service s_2 will influence the functioning of the service s_1. Also, more effort will be required to analyze and change this service. The changeability of the service s_2 will be also negatively influenced.

4.1.2.3 Direct extra-service relationships

Direct extra-service coupling covers the (direct) relationships between implementation elements belonging to different services. This is the worst type of coupling relationship and should be generally avoided.

If service s_1 has an direct extra-service relationship to service s_2, (an implementation element of service s_1 is calling directly an implementation element of service s_2), the analyzability, changeability and stability of the service s_1 will be strongly decreased.

For the analyzing this service, it is not enough just to analyze the contract (service interface) of service s_2, as it was the case for indirect extra-service relationship, but the internal element has to be analyzed directly. The functionality of the service s_1 will be influence even in the cases when the contract was not changed, which was also not the case of the indirect extra-service relationship.

4.1.3 Special types of coupling

Also, various special types of coupling in the context of service design has been discussed in [2].

4.1.3.1 Data coupling

Data coupling refers to the exchange of data parameters between software modules, e.g. passing a data parameter to a service operation. Data coupling can be minimized by only exchanging data which is necessary for the service operation to perform its task. Therefore analysis needs to be carried out to ensure that parameters exchange data that is necessary to implement the service operation. As the service interface constitutes a contract, careful attention needs to be paid to the minimization of data coupling at design time, avoiding unnecessary externalization of interface parameters.

4.1.3.2 Stamp coupling

Stamp coupling can occur in the situation where data is exchanged between services in the form of composite parameters as is the case in document-centric services that use complex XML message payloads. Stamp coupling refers to the case when the services are actually using only part of the composite parameter, each of the service using different part. It resulting in unnecessary coupling via externalized (optional) data structures In general, stamp coupling is an undesirable type of coupling and should be avoided. It may be acceptable in some cases where grouping of data elements significantly reduces interface complexity and the externalized data structure is not likely to change.

4.1.3.3 Control coupling

Control coupling refers to a situation where a service parameter controls the execution logic of the service by passing it information on what to do (e.g., passing a what-to-do flag)

This results in a poor design clarity, difficult maintenance, and reduction in service cohesion. Control coupling is regarded as highly undesirable with significant negative impact on service reusability.

4.1.4 Summary

In summary, to maximize reusability and maintainability, service design should aim to reduce coupling.

Although certain amount of intra-service coupling and indirect extra-service coupling is unavoidable in order to deliver required functionality, excessive coupling will create unnecessary complexity in the system. Special emphasis should be given on reducing data coupling and avoiding stamp and control coupling.

In chapter *5.1 Coupling metrics* we will look more closely on ways how we can measure different types of coupling using various metrics and predict possible problems early in the Software Development Lifecycle.

4.2 Cohesion

4.2.1 Definition

The cohesion of a service in a service-oriented system design is the measure of the degree to which the operations exposed in its service interfaces belong together conceptually [1].

It is defined on an ordinal scale and it is usually expressed as “high cohesion” or “low cohesion” when being discussed. Highly cohesive designs are desirable since they are generally easier to analyze and test, and provide better stability and changeability, which makes the system more maintainable.

Low cohesion has negative effects on analyzability, as modules with low cohesion are harder to understand. It takes longer time to developers have to go through longer parts of documentation to get the required information and it is harder to identify and repair defects, since they could be spread across many modules.

The changeability and stability of the system is negatively influenced as well, because changes in one module usually require changes in related modules. Also, logical changes in the domain affect multiple modules.

Also, such service is harder to reuse, because most applications won't need the random set of operations provided by the service.

4.2.2 Categories of Cohesion

Since the definition of cohesion is quite general, there are different interpretation of what does the “belongs together” mean.

Stevens et. al [7] proposed six categories of cohesion to be used for procedural paradigm: Coincidental, Logical, Temporal, Communicational, Sequential and Functional, each explaining the reasons why the procedures should be grouped together in the modules differently. These categories of procedural cohesion have been later redefined and extended by Eder et. al [6] in order to cover the conceptual and technological aspects introduced by the OO paradigm.

Finally these categories were once again redefined and extended in order to account for the distinguishing characteristics of SOC by Perepletchikov et. al in [16]. In total, eight categories of service cohesion were defined: Coincidental, Logical, Temporal, Communicational, External, Implementation, Sequential, and Conceptual.

These categories will be explained more in detail in the next chapters since they are essential for understanding the later discussed metrics.

4.2.2.1 Coincidental cohesion

According to the definition in [16] this category of cohesion occurs when "a service encapsulates unrelated functionality insofar as there are no semantically meaningful relationships between any of the operations exposed in its service interface."

This is the worst (or the weakest) type of cohesion. There is no real reason to group the operations of the service in such way since there is no relation between these operations. They are grouped coincidently. The only reason might be that they did not seem to logically belong anywhere else.

This type of cohesion is undesirable since it will negatively influence the analyzability of a service.

4.2.2.2 Logical cohesion

According to the definition in [16] this category of cohesion occurs when "service operations provide common functionality such as, for example, data update or retrieval."

It is important to say that service cohesion is considered as Logical type only if the logic based on which the operations are grouped together in the service is different that the logic used for latter categories of cohesion. Logical cohesion cannot be distinguished from Coincidental cohesion without semantic knowledge of the problem domain.

This type of cohesion is also very weak.

4.2.2.3 Temporal cohesion

When "service operations provide common functionality (as captured by Logical cohesion) and are performed within a predefined time period", the service cohesion is defined as temporal [16].

Temporal cohesion is defined as a more restricted version of Logical cohesion. Again, the temporal cohesion cannot be distinguished from Coincidental cohesion without semantic knowledge of the problem domain.

4.2.2.4 Communicational cohesion

For the communicational cohesion applies that "service operations operate on the same shared data abstractions." [16]

This type of cohesion assumes that if the operations of a service operate on the same data (e.g. a data representation of some business domain), they are doing similar thing. This kind of cohesion is desirable.

4.2.2.5 External cohesion

This was first introduced in [16] as a new category that was introduced in order to capture additional behavioral aspects of service interface cohesion, with definition that "service operations are used in combination by service consumers (clients).”

This category of cohesion is based on assumption that the fact that clients of a service uses more of the operations of the service reflects the cohesiveness of the service interface. The ideal state of this cohesion category is when all the consumers of a service uses all the operations of that service. This kind of cohesion is desirable.

4.2.2.6 Implementation cohesion

This category of cohesion was also first introduced in [16] and it is the case when “service interface operations are implemented by the same implementation elements.”

The Implementation category of cohesion was introduced in order to capture the aspects of service interface cohesion related to the underlying implementation of a service. It is also desirable.

4.2.2.7 Sequential cohesion

Sequential cohesion is when parts of a module are grouped because the output from one part is the input to another part like an assembly line (e.g. a function which reads data from a file and processes the data). According to the exact definition from [16], it is when “service operations are sequentially related insofar as either the output or post-condition from one operation serves as the input or pre-condition for the next operation.”

It is a strong form of temporal cohesion type and it is desirable.

4.2.2.8 Conceptual cohesion

The last category of cohesion is when “there is a meaningful semantic relationship between all operations of a service in terms of some identifiable domain level concept” [16].

The operations of a service contribute to either single business functionality or some other semantically meaningful concept such as an abstraction or data entity in the problem domain. This category represents the strongest type of service cohesion, given that it covers both the Functional and Model categories of classical and OO cohesion, which are considered to be the strongest categories of cohesion in the Procedural and OO paradigms respectively.

As such, it is highly desirable. However, note that the Conceptual category of cohesion is semantic in nature and is difficult to quantify by examining the structural properties of SO designs.

4.2.3 Summary

In this chapter, we have summarized the concept of cohesion. In order to increase software maintainability and reusability we should aim for conceptually cohesion.

In chapter *5.2 Cohesion metrics* we will look more closely on ways how we can measure different types of cohesion using various metrics and predict possible problems early in the Software Development Lifecycle.

4.3 Service Granularity

4.3.1 Definition

Service granularity reflects the the amount of functionality that is exposed by a service [citationneeded]. It reflects the degree of modularity of a system.

Generally we can speak about two types of granularity of services. Fine-grained services typically correspond to elementary business functions and implement highly reusable business logic. Coarse-grained services implement high-level business functions [2].

The benefits of fine-grained services include improved cohesion, reduction in coupling and better clarity of the design. However when designing services one must also consider the impact of using services over the internet and deal with the design constraints that this environment imposes. Such considerations include network latency and reliability and lead to a preference for coarse grained services that minimize the number of interactions needed to implement a given business function, reducing the complexity of the message interchange dialogue [1].

The recursive nature of service composition allows the construction of business level services from reusable fine-grained services. Business-level services can then be made available to external client applications (i.e. external to the enterprise or administrative domain), while fine-grained services are used within the enterprise taking advantage of greater flexibility and reuse [11].

4.3.1.1 Service Granularity Classification

[11] provides further classification of service granularity based on three different interpretations of the term.

Functionality (or capability) granularity refers to how much functionality is offered by a service. Secondly, data granularity reflects the amount of data that is exchanged with a service. Finally, the business value granularity of a service indicates to which extent the service provides added business value.

4.3.2 Summary

In this chapter, we have summarized the concept of service granularity. Service granularity se determined by the overall quantity of functionality encapsulated by a service.

Several trade-offs need to be considered when choosing the correct level of service granularity. Fine-grained services typically leads to better reusability of the system, while coarse-grained services can lead to better network latency and reliability of the system. However, service composition can help with leveraging advantages of both.

In chapter *5.3 Service granularity metrics* we will look more closely on ways how we can measure service granularity using various metrics and predict possible problems early in the Software Development Lifecycle.

4.4 Reusability and composability

4.4.1 Definition

Although the concept of reusability was already mentioned in the quality characteristics section, it can be considered as a structural characteristic as well [4].

A service should ideally be designed for more that one service consumer. Service composability is a form of reusability. A service becomes a composition participant and can be reused along with other services to provide business functionality.

Sindhgatta et al. [4] outlines two perspectives reusability of an entity may be looked at from: the characteristics of the entity that are predictors of reusability, and potential for future reuse of the entity based on usage that has already happened. The attributes of coupling and cohesion are generally good predictors of reusability. A service whose operations are cohesive and have less external dependencies will be more easily reusable.

4.4.2 Summary

In this chapter, we have summarized the concept of reusability and composability.

In chapter *5.4 Metrics for predicting Reusability* we will look more closely on ways how we can predict reusability using various metrics.

5 Service Design Metrics

Metrics can be used as early predictors of the maintainability quality characteristic of service oriented software systems.

5.1 Coupling metrics

5.1.1 Coupling metrics by Perepletchikov et al. [16]

The most sophisticated suite of metrics was proposed by Perepletchikov et al. in [16]. In this paper eleven primary metrics and additional five aggregated metrics were proposed.

5.1.1.1 Primary metrics

Primary metrics provides low level of abstraction. They are measuring concrete types of coupling relations and can be used for detailed analysis of different service coupling notions. However, for more high-level view on level of coupling inside system, usage of aggregate metrics discussed in the next chapter is more suitable.

5.1.1.1.1 Weighted Intra-Service Coupling Between Elements

This metric measure the intra-service coupling between design elements belonging to the same service. Weighted Intra-Service Coupling Between Elements (WISCE) for given element equals to the weighted count of the number of other implementation elements of the same service to which the element is coupled via incoming or outgoing relationships.

It is based on an assumption that this kind of coupling should be avoided, because the impact of high coupling between these elements on maintainability is expected to bu similar to that suggested for the Procedural or OO systems. High values of WSICE metric will result in decreased analyzability and changeability of the service.

5.1.1.1.2 Service Interface To Intra Element Coupling

Service Interface To Intra Element Coupling (SIIEC) metric basically look for the number of implementation elements that directly implements the interface of the service. More specifically, SIIEC for a given service is a count of the relationships between its service interfaces and the implementation elements belonging to the service that directly implement operations exposed in the interface as part of the intra-service coupling.

A large number os service implementation elements invoked from a service interface can result in the decreased analyzability of the service due to an unnecessarily tight linkage between its interface and implementation.

It is advised to keep the SIIEC values as close to one as possible. This can be achieved by having a dedicated implementation element to which all service operations are initially mapped.

5.1.1.1.3 Extra-Service Incoming Coupling of Service Interface

Extra-Service Incoming Coupling of Service Interface (ESICSI) metric look for the for the number of elements that are calling the service through the service interface. ESICS for a given service S is a count of the number of system elements not belonging to service S that couple to this service through its interface as part of the indirect extra-service coupling.

This kind of coupling is unavoidable in practice since the services in Service-Oriented Computing should communicate with one another via interfaces in order to achieve some desired functionality, thus it is considered as the desirable form of (loose) coupling. But if this coupling is excessive, it can negatively influence changeability and stability of a system.

5.1.1.1.4 Element to Extra Service Interface Outgoing Coupling

Like the ESICSI, Element to Extra Service Interface Outgoing Coupling (EESIOC) metric is based on similar assumptions about indirect extra-service coupling, but this metric focuses on outgoing part of the coupling instead. EESIOC for a given service implementation element is a count of the number of other service interfaces to that are used (coupled to) by this implementation element as part of the indirect extra service coupling.

It is based on the assumption that high outgoing indirect extra-service coupling from a given service implementation element will negatively influence the analyzability, changeability and stability of that element, since there is a possibility that changes to the operations exposed in service interfaces will influence its functioning (stability), and also more effort will be required to analyze and change this element.

5.1.1.1.5 Weighted Extra-Service Incoming Coupling of an Element

Weighted Extra-Service Incoming Coupling of an Element (WESICE) metric look for direct extra-service coupling, that means for relationships between implementation elements belonging to different services. WESICE for a given service implementation element of a particular service is the weighted count of the number of system elements not belonging to the service that are coupled to (use) this implementation element as part of the direct extra-service coupling.

This type of coupling can be considered as the worst type of extra-service coupling coupling and thus should be avoided. This is because the direct extra-service relationships result in explicit dependencies between implementation of services, thereby decreasing the reusability of such. It also negatively influences changeability and stability of the system.

5.1.1.1.6 Weighted Extra-Service Outgoing Coupling of an Element

Weighted Extra-Service Outgoing Coupling (WESOCE), similarity to the WESICE looks for direct extra-service coupling, but just in the other direction. By the definition, WESOCE for a given service implementation element of a particular service is the weighted count of the number of system elements not belonging to the same service that are used by this element as part of the direct extra-service coupling.

High outgoing direct extra-service coupling will negatively influence the analyzability, changeability and stability of element.

5.1.1.1.7 Number of Coupled Incoming Services

Number of Coupled Incoming Services (NCIS) is a variation on ESICSI and WESICE metrics discussed earlier. NCIS for a given service is a distinct count of other services in the system having elements connecting to service to this service through either its implementation elementary (direct extra-service coupling) or its service interface (indirect extra-service coupling).

However, there are two differences between NCIS and the ESICSI/WESICE metrics:

1. NCIS measures coupling at the service level without considering the individual couples between service interfaces or implementation elements as was the case with ESICSI and WESICE metrics; and
2. NCIS does not differentiate between the direct and indirect extra-service relationships.

Such differences can be considered as the limitation of this metric since they can potentially result in the loss of measurement accuracy. However, NCIS can especially useful for quantifying the coupling between „black box” services during the Analysis phase of SDLC after all major services in the system have been identified but not yet designed in terms of the concrete implementation elements.

5.1.1.1.8 Number of Coupled Outgoing Services

Number of Coupled Outgoing Services (NCOS) is an “outgoing” version of NCIS metric. NCOS for a given service is a distinct count of other services in the system, to which this service is coupled either through an element (direct extra-service coupling) or a service interface (indirect extra-service coupling).

This metric is a variation of the EESIOC and WESOCE metrics, with same differences as before. Also, similar relation to quality characteristic applies here.

5.1.1.1.9 System Partitioning Factor

System Partitioning Factor (SPARF) for a given Service-Oriented system measures the degree of partitioning of this system into services. More specifically, SPARF is the ration of total elements in the system belonging to at least one service to the total number of the elements in the system.

Values of SPARF will range from zero to one, where zero means that all the elements in the system belongs to at least one service. Conversely, a value of zero indicates the total absence of services in the system.

A high number of elements that do not belong to any of the system services will result in decreased analyzability of a system due to a lack of system modularization, and will also undermine one of the core principle of service-orientation that a system should be constructed as a set of interacting services.

5.1.1.1.10 System Purity Factor

System Purity Factor (SPURF) for a given Service-Oriented system measures the degree of purity of this system in terms of all implementation elements belonging to one and only one service. More specifically, SPURF is the inverted ratio of the number of intersected services to the total number of services in the system.

A high number of services implementation elements that belong to more than one service can increase the interdependencies between different services, consequently decreasing their reusability and breaking the principle of service-autonomy, which in turn influences most sub-characteristics of of maintainability. For example, the stability of a system will decrease since changes to the shared implementation elements can influence more than one service. Moreover, it will be difficult to physically 'relocate; a service (deploy it to different hardware or software environments) comprised of shared implementation elements without affecting the other services in the system that share these elements, thereby reducing system changeability.

5.1.1.1.11 Response for Operation

Response for Operation (RFO) is the only metric from this set that actually measures dynamic coupling. It is defined for given operation as the total number of set elements of service implementation elements and service interfaces that can be potentially invoked (or executed) in response to the invocations of this operation with all possible inputs.

A high number of design elements interacting in order to achieve some desired functionality in response to the invocation of an operation will result in the decreased analyzability of a system since the entire call chain needs to be analyzed in order to understand the functioning of the operation. Also, the stability will be affected since an element to which a given operation belongs will be dependent on an increasing number of external elements.

5.1.1.2 Aggregation metrics

The purpose of the aggregation metrics is to combine together the relationship-based metrics defined in the previous section in order to support the quantification of coupling at a higher level of design abstraction.

5.1.1.2.1 Total Weighted Intra-Service Coupling of a Service

Total Weighted Intra-Service Coupling of a Service (TWISC) for a given service is a sum of all intra-service related measures for each of its implementation elements, combined with the measure of coupling between its service interface and implementation elements that directly implement this interface.

TWISC measures the total intra-service coupling of a service based on:

1) intra-service coupling of all its implementation elements (WISCE metric), plus
2) intra-service coupling of its interface (SIIEC metric).

Services are intended to be independent components and thus be maintained in isolation from the system. Therefore, it is useful to measure the total coupling within a single service. A high value of TWISC can indicate bad internal design structure of a particular service and can lead to low maintainability.

5.1.1.2.2 Total Weighted Incoming Extra-Service Coupling of a Service

Total Weighted Incoming Extra-Service Coupling of a Service (TWIESC) for a given service is a sum of all indirect (via service interface) and direct (between implementation elements) incoming extra-service measures for its constituent elements.

TWIESC measures the total incoming extra-service coupling of a service based on:

1) indirect extra-service incoming coupling of interface (ESICSI metric)
2) direct extra-service incoming coupling of its implementation elements (WESICE metric)

TWIESC quantifies the dependency of the rest of the system on this service, thereby providing an indication of how critical the service is within a system wide context. To this end, this metric can be used as the indicator of service changeability.

5.1.1.2.3 Total Weighted Outgoing Extra-Service Coupling of a Service

Total Weighted Outgoing Extra-Service Coupling of a Service (TWOESC) for a given service s is a sum of all indirect (via service interface) and direct (between implementation elements) outgoing extra-service measures for its constituent elements. TWOESC measures the total outgoing extra-service coupling of a service based on:

1) indirect extra-service outgoing coupling of its implementation elements (EESIOC metric)
2) direct extra-service outgoing coupling of its implementation elements (WESOCE metric)

TWOESC quantifies the dependency of a service on the other services in the system, and as such, it can be used as the indicator of service stability.

5.1.1.2.4 Total Weighted Coupling of a Service

Total Weighted Coupling of a Service (TWCS) for a given service is a combination of all the service-level coupling measures for this service.

This metric quantifies the overall coupling of a service based on all possible types of coupling. In particular, the following coupling aspects are measured:

1) intra-service coupling (TWICS metric)
2) incoming indirect/direct extra-service coupling (TWIESC metric)
3) outgoing Indirect/direct extra-service coupling (TWOESC metric)

TWCS can be used to quantify the quality of the internal design structure of a service, the criticality of a service within a system wide context and the degree of dependency of a service on the other services in the system.

5.1.1.2.5 Response for Service

Response for Service (RFS) for a given service is the sum of the RFO (Response for Operation) measures for each of the operations exposed in its service interface.

RFS quantifies the dynamic coupling of a service as reflected by the number of service implementation elements and other service interfaces that can be potentially invoked in response to all possible invocations of all operations exposed in the interface of this service. A large number of internal/external service implementation elements and other service interfaces included in the collaboration sequences of service will have a negative effect on the analyzability and stability of this service due to the strong dependency on a large number of other elements in the system.

5.1.1.3 Summary and discussion

Perepletchikov proposed a very complex set of metrics for measuring coupling at early stages of the Software Development Lifecycle. The author defines metric for each type of the relationship as discussed in chapter *4.1.2 Coupling relationships,* and provides also a set of aggregate metrics to give an overview of service coupling.

Most of those metrics are measuring static coupling, thus are usable for measuring coupling in design phase of the Software development Lifecycle.

However, there are several disadvantages. First, these metrics are measuring coupling based on the number of relationships between design elements an type of those relationships, they do not take into account the strength (or nature) of given relationship.

Also, internal implementation details of the services need to be known in order to use these metrics. It is arguable whether the additional level of detail offers newer insight to the problem.

From the practical point of view, the biggest problem might be the absence of a tool for automatization of the measurement procedure. Measuring the whole set of metrics manually might be quite time consuming, especially for complex service oriented systems.

Unfortunately, these metrics have not been empirically evaluated so the practical applicability is yet to be determined.

5.1.2 Coupling metrics by Sindhgatta et al. [4]

The next set of metrics we will be discussing was introduced in [4]. The focus of the author was on defining a small set of metrics that would be easy for the service designer to act on and the service consumer to comprehend.

5.1.2.1 Service Operational Coupling Index

Service Operational Coupling Index (SOCI) analyze the dependence of a service on the operations of other services it uses for its functionality. It equals to the number of operations of other services invoked by given service.

SOCI is an adaptation of the OO metric Response for a Class (RFC), in the services domain. It is identical with the Response for Service (RFS) metric proposed by [16].

This metric can detect problems, that could have negative effect on the analyzability and stability of the system.

5.1.2.2 Inter-Service Coupling Index

Inter-Service Coupling Index (ISCI) is defined as the number of services invoked by a given service. It is very similar to SOCI metric, only operating on different level of abstraction (using a number of services instead of a number of operations).

The same relations to quality characteristic apply here.

5.1.2.3 Service Message Coupling Index

Service Message Coupling Index (SMCI) metric measures the dependence of a service on the messages derived from the information model of the domain (i.e. messages that service operations receive as inputs, and produce as output via the declared interface).

A low SMCI indicate less complexity for the service in interpreting and creating messages and less dependence on the domain model, thus better analyzability and stability.

5.1.2.4 Summary and discussion

Sindhgatta et al. proposed much simpler set of metrics for measuring service coupling. Used together, the metrics provide enough detail on service coupling and can be very useful for identifying potentially problematic services.

Again, these metrics are measuring static coupling. Moreover, they do not require internal implementation details about services, which is an advantage in certain situations (e.g. when we want to use these metrics very early in the Software Development Lifecycle when these details are not yet available).

5.1.3 Coupling metrics by Pham Thi Quynh and Huynh Quyet Thang [10]

Pham Thi Quynh and Huynh Quyet Thang in [10] proposed four new service coupling metrics.

These metrics are different from previous metrics in the fact that they are dynamic. They are focused on interaction between services in a system at runtime to bring more exact results.

5.1.3.1 Coupling Between Services Metric

The Coupling Between Services Metric (CBS) is looking for the number of relationships between a service and all other services in system. It is similarity to ISCI metric proposed in [16].

Higher value of CBS metric for service means tighter the relationship with other services, and thus lower maintainability.

5.1.3.2 Instability Metric for Service Metric

The Instability Metric for Service Metric (IMS) is adapted on formula for calculating instability of a component based on fan-in and fan-out metrics, that was proposed by Joost Visser in [8].

This metric shows interaction between a service and others in system through sending and receiving messages. It is computed as a percentage of messages that are sent by a service from the total amount of the messages that are sent or received by that service.

If the value of this metric is low, the level of dependency of service is low whereas others depend on it higher. IMS = 100 % means that the service is only sending messages, and thus according to the metric it is very instable, IMS = 0 % means that the service is only receiving messages and thus the stability is very high.

5.1.3.3 Degree of Coupling between 2 services metric

The Degree of Coupling between 2 services metric (DC2S) is developed from CBS, but instead of looking on all the services in the system, it focuses on relationship only between two services to detect the dependency between these services.

Considering service s_1 and service s_2, the DC2S metric between s_1 and s_2 is calculated by the percentage of the number of times from s_1 to s_2 in the number of times from s_1 to other services in system.

DC2S metric identifies the level of coupling between two services in runtime; for example, in specification service s_1 has relation to service s_2 and service s_3. However, in runtime, s_1 calls to s_2 by 100 times, whereas it only calls to s_3 by 3 times. This shows that service s_1 couples with service s_2 tighter than service s_3. From this point, when maintaining service s_1, we should concentrate the level of impact of service s_2 higher than service s_3.

5.1.3.4 Degree of Coupling within a given set of services metric

The last metric proposed in [10] is called Degree of Coupling within a given set of services metric (DCSS). This is a metric that use graph representation of the services (represented as nodes in the graph) and their interactions (represented as edges in the graph). In this graph, the edges have the direction of the interaction between services and have weight of the edge is the number of times the one service makes the request to the other. The ability of coupling of a service is is defined as the level of easy to reach a node in the graph.

5.1.3.5 Summary and discussion

Above discussed set of metrics is measuring dynamic coupling of service, thus it has to be used later in the Software Development Lifecycle as the system needs to be already implemented.

On the other hand, it can provide more precise results, and thus it can be very useful during the maintenance phase of the Software development Lifecycle.

5.1.4 Coupling metrics by Feuerlicht [3]

This metric is measuring the quality of service design based on orthogonality of services. It is using the level of data coupling between services as an indirect indication of orthogonality.

5.1.4.1 Data Coupling Index

Data Coupling Index (DCI) is defined as the average number of shared schema elements for each interface message pair combination.

High values of DCI indicate low orthogonality of service design with corresponding negative impact on reuse and adaptability. Lack of orthogonality of a set of services involves duplication of functionality and data structures and is associated with high levels of coupling and low levels of service cohesion, reducing the potential for reuse.

Author provides also another view on this problem using a normalized DCI index (NDCI), defined as the ratio of DCI over ANCT (Average Number of Complex Types per interface). The value of NDCI represents the average number of top level complex elements that are shared with other interfaces.

High values of NDCI again indicate a high level of stamp coupling.

5.1.4.2 Summary and discussion

This could be an extremely useful metric, especially when used in domain-wide document-centric SOA environments.

The fact that the metric is very easy to interpret is also an advantage.

In the chapter *6. Tool for evaluating a service design metrics based on coupling between service interfaces* this metric will be discussed more deeply with practical demonstration on real-life data.

5.2 Cohesion metrics

Cohesion is considered to be one of the most difficult to measure structural properties of software due to its inherently semantic nature. Also, the current empirical understanding of the notion of cohesion is not as advanced as the understanding of other structural properties such as coupling and complexity

5.2.1 Cohesion metrics by Perepletchikov et al. [5]

The first cohesion metrics that took into consideration the service interfaces were proposed in the work of Perepletchikov et al [8]. In this research the authors consider various notions of cohesion and propose following metrics.

5.2.1.1 Service Interface Data Cohesion

The Service Interface Data Cohesion (SIDC) metric was designed to directly quantify the Communicational cohesion category, as well as indirectly reflect the Coincidental and Conceptual categories of cohesion.

The SIDC metric quantifies cohesion of a given service based on the cohesiveness of the operations exposed in its service interface, as reflected by all service operations:

- having common parameter types
- having the same return type

A service is considered to be communicationally cohesive when all of its service operations share (or use) common parameter and return types.

5.2.1.2 Service Interface Usage Cohesion

Service Interface Usage Cohesion (SIUC) concentrate on the cohesion between the customers of the service and the service itself. The value of the metric is the number of service consumers that use all the operations.

The SIUC metric quantifies the usage patterns of service operations, thereby it is directly related to the External category of cohesion, as well as indirectly influencing the Coincidental and Conceptual categories of cohesion.

A service is deemed to be Externally cohesive when all of its service operations are invoked by all the clients of this service.

5.2.1.3 Service Interface Implementation Cohesion

This metric focus on measuring the cohesion between a service interface and the internal service implementation. The values of this metric is calculated as a function of the number of operations that use the same internal implementation elements.

The Service Interface Implementation Cohesion (SIIC) metric covers the implementation features of service operations, thereby being directly related to the Implementation category of cohesion, as well as indirectly influencing the Coincidental and Conceptual categories.

5.2.1.4 Service Interface Sequential Cohesion

The Service Interface Sequential Cohesion (SISC) metrics quantifies sequential properties of the usage patterns of service operations. Service interface is cohesive, if its operations are sequentially dependent in the sense that the types of certain output parameters of one operation match the types of certain input parameters of another operation.

It is being directly related to the Sequential category of cohesion, as well as indirectly influencing the Coincidental and Conceptual categories of cohesion.

A service is deemed to be Sequentially cohesive when all of its service operations have sequential dependencies, where a post condition/output of a given operation satisfies a precondition/input of the next operation. As with the SIUC metric, the SISC metric is associated with the communication (usage) pattern of service operations. The difference is that in the case of SISC, the sequential dependencies between service operations are also taken into consideration.

5.2.1.5 Total Interface Cohesion of a Service

The Total Interface Cohesion of a Service (TICS) metric covers all possible aspects of service interface cohesion as captured by the previously defined metrics, thereby quantifying the total (overall) cohesion of a service. Specifically, TICS quantifies cohesion of a service based on the following characteristics of its service interface:

1. service operations having common parameters and return types (SIDC metric)
2. service operations being invoked by every client of the service (SIUC metric)

3. service operations being implemented by the same service implementation elements (SIIC metric)
4. service operations having sequential dependencies (SISC metric)

To this end, TICS can potentially suggest the best possible cohesiveness of a service (the Conceptual cohesion category), or a total lack of cohesiveness (the Coincidental category).

5.2.1.6 Summary and discussion

The limitation of these metrics metrics is that operations which operate on data characterized by similar, but not exactly matching, types are treated as being totally unrelated. Such cases of operations are frequent in real world services. These metrics thus may overestimate the cohesion lack of service interfaces.

Unfortunately, these metrics have not been empirically evaluated.

5.2.2 Cohesion metrics proposed by Sindhgatta et al. [4]

Three metrics for measurement of cohesion were proposed in [7]. First two of them (LCOS1 and LCOS2) are based on widely accepted LCOM metric, that is used as a measure of cohesiveness in OO system, the third is completely new metric.

5.2.2.1 Lack of Cohesion of Service Operations (LCOS1 and LCOS2)

For the Lack of Cohesion of Service Operations metrics, author assumes that if service operations use common messages or their constituent data types, they can be considered cohesive.

The LCOS1 metric looks for the number of operation pairs within a service, that share some of the messages. If the number of operation pairs that share messages is more that the number of pairs that do not, the service is considered to be strongly cohesive (LCOS1 = 0). Otherwise, the difference between the number is taken as the lack of cohesion measure.

The LCOS2 metric is looking for the average number of operations that use each message and the total number of operations.

In the ideal case, when all the message is used by each of the operations, the service is considered to be highly cohesive (LCOS2 = 0). In the opposite case, if each operation uses a distinct message, then the service is considered not to be cohesive (LCOS2 = 1).

5.2.2.2 Service Functional Cohesion Index

5.2.2.2 Service Functional Cohesion Index (SFCI) is based on an assumption that a cohesive service typically operates on a small set of key business objects (messages) relevant to that service. These objects should appear in most of its operations.

So SFCI metric is looking for a message that is reused across the operations within a service the most. SFCI equals to the ratio of the number of operations using this message to the total number of operations in the service.

5.2.2.3 Summary and discussion

According to the empirical evaluation, both LCOS1 and LCOS2 suffer from some drawbacks when applied to service oriented systems [4]. First, the discrimination power of LCOS1 is low, and most service tend to be classified as highly cohesive.

LCOS2 tends to increase sharply with increase in the number of operations, and most services appear as lacking cohesion. This is because, with an increasing number of operations, it becomes very unlikely that each operation will require the same se of (all) messages, although they may still contain some core data types that are relevant to the service functionality and may thus be argued to be functionally cohesive.

These metrics need significant improvements in order to be used effectively.

5.2.3 Cohesion metrics by D. Athanopoulos and A. V. Zarras [9]

Another set of metrics was proposed by D. Athanopoulos and A. V. Zarras in [9]. The authors here points out the problem with definition of metrics measuring sequential and communication cohesion proposed in [16] – there the message similarity was defined only as boolean value. Messages could be either similar (the parameter types equal), or not.

As a reaction, two fine-grained metrics of cohesion lack are proposed, with respect to the structural similarity of the input/output data types of interface operations.

A message is modeled as an unordered rooted tree. The tree root represents the message. The non-leaf vertices correspond to complex elements, i.e. elements characterized by a name and a complex XML type, which consist of further constituent elements. The leaves of the tree represent primitive elements, i.e. elements characterized by a name and a XML build-in type.

The message similarity is redefined as the number of the bottom-up subtrees that the messages have in common divided by the order of the message that results from the union of those two messages. The values of message similarity range from 0 (meaning that the messages are completely unrelated) to 1 (the messages exactly match).

Also, a special attention is paid to certain common elements, that are not related to the particular functionality of the operations, but are rather used by the infrastructure. These elements are excluded when computing the metrics values.

5.2.3.1 Lack of sequential cohesion metric

Lack of sequential cohesion metric (LoCs) is looking for the lack of sequential cohesion, which is defined as the complement of the average sequential similarity of the pairs of operations that belong to given service.

The sequential similarity between two operations op_i, op_j of an interface si is defined as the average of:

4) the similarity of the input message of op_i and the output message of op_j, and

5) the similarity of the output message of op_i and the input message of op_j

An interface of a service is sequentially cohesive to some extent, if it includes pairs of operations op_i, op_j, such that the input message of op_j (resp. op_i) and the output message of op_i (resp. op_j)

comprise common (complex or primitive) elements. More specifically, for complex elements the term common refers to elements characterized by the same complex XML type, while for primitive elements the term common refers to elements, characterized by the same name and build-in type.

In this case, the operations op_i, op_j are sequentially related, in the sense that certain output data produced by op_i (resp. op_j) may be used as input for op_j (resp. op_i).

5.2.3.2 Lack of communicational cohesion metric

The lack of communicational cohesion is defined as the complement of the average communicational similarity of the pairs of operations that belong to given service.

The communicational similarity between two operations op_i, op_j of an interface si is defined as the average of:

1) the similarity of the input message of op_i and the input message of op_j, and
2) the similarity of the output message of op_i and the output message of op_j

An interface is communicationally cohesive to some extent, if it includes pairs of operations op_i, op_j, such that their input message and/or their output messages comprise common (complex or primitive) elements. In this case, the operations are communicationally related, in the sense that they may use similar input data and/or produce similar output data.

5.2.3.3 Summary and discussion

Metrics discussed above provides very convenient way how to measure sequential and communicational cohesion, as they provide better detail on the messages. They fix problem of other metrics, that are considering almost-similar messages as completely unrelated. Thus the results are more precise and useful.

5.3 Service Granularity Metrics

In this chapter, we will review available metrics for measurement service granularity.

5.3.1 Granularity metrics proposed by Sindhgatta et al. [4]

Here, the authors propose several metrics for measuring capability and data granularity on different level of abstraction.

5.3.1.1 Service Capability Granularity and Service Data Granularity

First, authors looked on service level of abstraction and took size of the service as the indicator for capability granularity and amount of data transferred to provide that functionality as the indicator for data granularity.

Specifically, Service Capability Granularity (SCG) metric equals to the number of operations in service, whereas Service Data Granularity (SDG) metric equals to the number of messages used by these operations. Higher values may indicate coarser granularity, e.g. larger functional scopes.

However, this information alone is not enough to measuring service granularity. Capability granularity of each service operation is equally important. For example, if we start to decomposing coarser operations into multiple finer-grained, the value of SCG could change significantly, even though the the functionality offered by the service did not change (and thus the service capability granularity should not change either according to the definition).

Unfortunately, measuring the amount of functionality provided by single operation could be especially difficult. Authors are trying to solve this problem by measuring the granularity at the process level as well.

5.3.1.2 Process Service Granularity, Process Operation Granularity and Depth of Process Decomposition

Here the authors assume that too many services and operations constituting a business process may imply that the service in the design model are too fine grained, and that there is a need to re-factor the services to get the granularity right.

Process Service Granularity (PSG) equals to the number of services invoked by a business process. Conversely, Process Operation Granularity (POG) equals to the number of operations invoked by a business process.

However, several levels of composition might occur in the service design. Another metric proposed to cover this is the Depth of Process Decomposition (DPD), which equals to the number of levels to which the process was decomposed before service were identified.

5.3.1.3 Summary and discussion

It can be hard to understand the outcome of the measurement with the metrics discussed above. First of all, they need to be used together to obtain any reasonable data. But not guidelines how to combine those results were proposed.

From this point of view, the metric doesn't seem to be very useful for the first-time measurement for instant problem identification (as the optimal service granularity depends on several other aspects anyway), however, they can be useful for checking whether the level of granularity is constant, or if it is changing to one or the other direction over time.

5.4 Metrics for predicting Reusability

5.4.1 Reusability metrics proposed by Sindhgatta et al. [7]

Sindhgatta et al. proposed a set of metrics for predicting service reusability based on the current level of reuse.

5.4.1.1 Service Reuse Index

The Service Reuse Index (SRI) metric looks for the number of existing consumers of a service. It assumes that higher the number of existing consumers of a service, the higher the probability for future reuse of this service is. At the service design level, these consumers may be other services coupled to this service or business processes where the service is used.

5.4.1.2 Operation Reuse Index

The Operation Reuse Index (ORI) for an operation is the number of consumers of that operation across service and business processes. This is a variation of Service Reuse Index, with the detail on operations.

5.4.1.3 Service Composability Index

The Service Composability Index (SCOMP) looks for the compositions in which the service is a composition participant and the number of distinct composition participants which succeed or precede the service.

It assumes that the higher the number of distinct composition participants which succeed or precede the service, the higher the probability for future reuse of this service is.

5.4.1.4 Summary and discussion

This set of metrics for measuring reusability proposed by Sindhgatta et al. simple and easy to use, however the practical usefulness of these metrics is questionable. The level of use is usually connected not only with the structural properties of the service design, but with the business domain as well, and thus it might be very hard to interpret and act on the results of these metrics. Nonetheless, in combination with other metrics these metrics can provide another view on the problem.

6 Tool for evaluating a service design metrics based on coupling between service interfaces

In order to use the metrics effectively, tools for automatization of the measurement procedure are essential as the measurement procedure is usually very complicated and the data that need to be analyzed are quite complex. Ideally, these tools would be directly integrated with the tools the SOA architects are already using today and would be using data that are already available (e.g. UML diagrams in XML format or WSDL files).

In this chapter, an implementation of a tool for computation of the DCI metric will be demonstrated and the DCI metric will be evaluated using this tool on real-life data.

6.1 Goals overview

The goal is to create a tool for software architects that are designing service interfaces within service oriented architecture. The tool should be used as a helper for detecting problems in the service interfaces (message structures). The tool should be using data that are already available in order to be used effectively. In our case, it will be using WSDL definition in combination with XSD schema files as a input for measuring the DCI.

In order to guide the software architects, the tool should be able to:

- Compute the DCI according to the metric,
- Point out problematic operation interfaces, that use most of the shared complex types,
- Compute the NDCI according to the metric,
- Point out problematic complex types, that are reused the most.

The expected advantages of using this tool over computing the metric manual way are following:

- Faster results, as the computation will be automatic,
- More precise results, as the computation won't be limited only to complex elements from the top level,
- Additional help with locating problematic operations and complex elements so the corrections can be made easily.

6.2 Implementation

The tool will be implemented as a standalone command line application. In this chapter the internal details of the tool will be briefly presented. The source code and the executable of the application can be downloaded from a public source code repository on GitHub:

https://github.com/hejld/DCI-Metric

The tool is using WSDL files as input for loading the operation interface definitions. EasyWSDL library was used for loading the WSDL and XSD definitions. Both WSDL version 1.1 and 2.0 can be used as an input for the metric tool. However, several bugs were found in this library during the implementation (e.g. the library does not expect <xs:sequence> or <xs:choice> element inside another <xs:sequence> or <xs:choice> element, although it is perfectly valid). In order to load the definitions correctly, few workarounds were used.

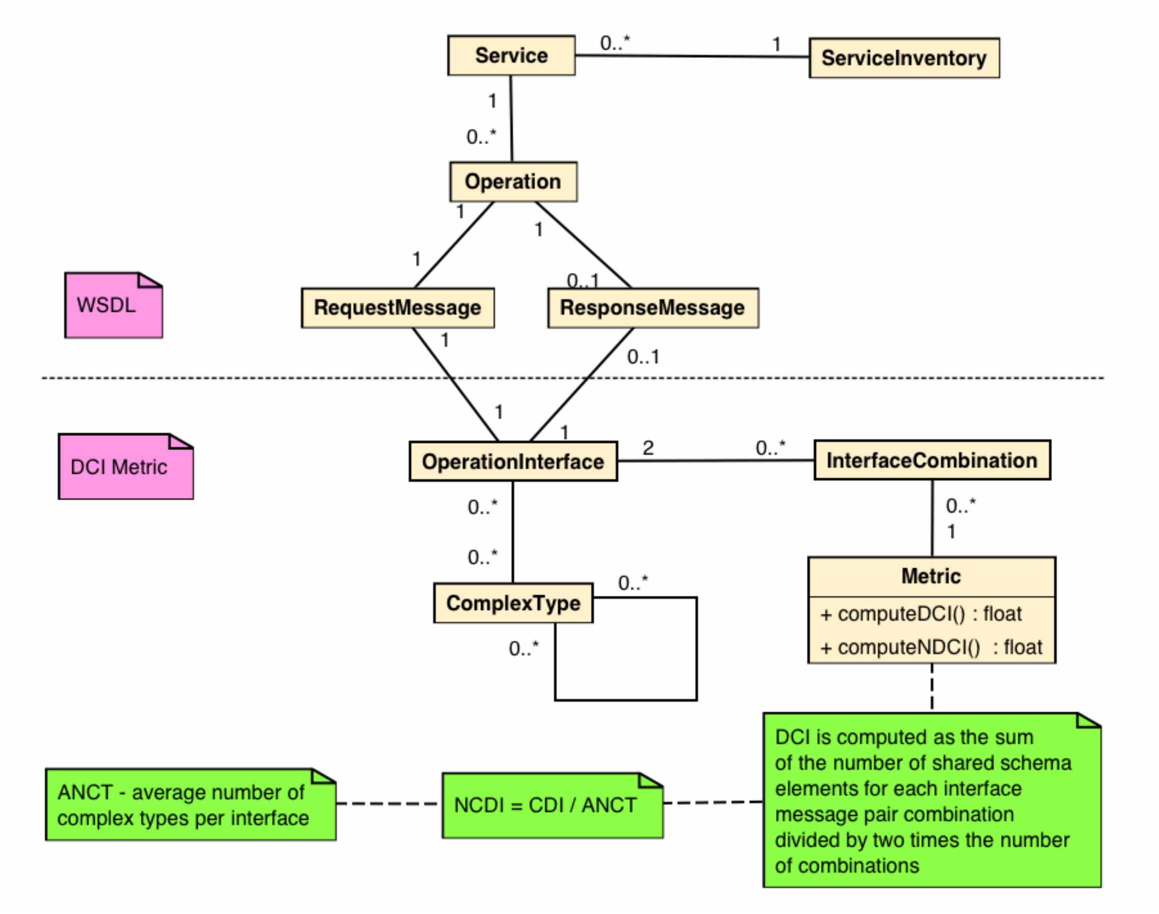

Illustration 4: Conceptual view on the tool, source: author

Several WSDL definitions can be loaded to be used for the metric computation. From each of the WSDL definition, all the services defined are analyzed and all the service operations are loaded, together with sets of messages that define the interface of those service operations. For each of the operations, a set of all distinct named complex types that are used in the operation messages is prepared. Also, the total number of complex types used in the each interface is counted.

During the metric calculation, all the possible combinations of non-equal interfaces are prepared and for each of those combination a list of complex types that are used in both interfaces is prepared.

DCI is computed as the sum of numbers of shared complex types for all the operation combinations divided by the two times the number of combinations. NDCI is computed as DCI over the average number of complex types per interfaces (ANCT).

The tool has an intuitive command line interface that guides the user throughout the process.

6.3 Evaluation of the tool and DCI Metric using OTA Air Messages

In this chapter, we will evaluate the DCI metric in the similar manner as was originally evaluated in [3], using the OTA Air Message definitions [18]. However, in this case the results will be slightly different as we won't be restricted only to the top level complex elements.

First, lets look on the original results of the metric from [3].

DCI = 3.56,

ANCT = 11.67,

NDCI = 0.30.

Now, we will compare them with the new result obtained using the tool.

DCI = 4.348485,

ANCT = 30.666666,

NDCI = 0.14179842.

The difference between the results reflects the shared complex types on lower levels. In average, only about 0.23 of complex type per interface combination was shared on other than top level. This also affected the NDCI, which with is now significantly lower. Now in average only about 14.18% of complex type were shared between the interfaces per interface combination.

Also additional information provided by the tool is the average number of shared complex schema elements per interface and number of complex types that were actually reused.

Average number of shared complex schema elements per interface is about 6.5, which generally means that in average about 21% of the complex elements of each interface are actually used on more than one place. There are totally 14 of complex types that were reused across the messages.

6.3.1 Detailed analysis of shared complex types

Using the advanced features of the tool, more detailed analysis of the shared complex types can be made.

Complex types	**Usage counts**
POS_Type	12
WarningsType	11
SuccessType	11
ErrorsType	11
FreeTextType	8
UniqueID_Type	5
LocationType	4
CompanyNameType	4
EquipmentType	2
AirItineraryType	2
OperatingAirlineType	2
FareType	2
PersonNameType	2
SpecificFlightInfoType	2

Table 1: Detailed analysis of shared complex types, source: author

As we can see from Table 1, the most used complex types were *POS_Type*, which was used by every interface, and *WarningsType*, *SuccessType* and *ErrorsType*, each with 10 usages.

With this information, it is easier to decide if such data structures duplication represents a threat for service reusability.

According to the documentation annotation of the *POS_Type*, this element provides information on the source of a request. As we can see, this complex type is related rather to the infrastructure, than to the business function of the service. The same applies for *WarningsType*, *SuccessType* and *ErrorsType* complex types, which has rather simple structures (*SuccessType* even does not have any content at all), and are used in all response messages to reflect the response status of the messages.

From this point of view, I would not consider the reusing of those data structures as potential problem for service reusability as suggested by the metric.

Another complex type with high usage count, *FreeTextType*, also does not represent any threat to the reusability, as it is only a very simple extension to the *xs:string* simple type, containing additional information about the language of the text. This is also not related to the business function of the service and does not represent any threat to the reusability.

As we can see, out of the total of 78 usages of the shared elements, 53 are of the infrastructure elements and only 25 are really connected to the business domain of the services.

6.3.2 Detailed analysis of operation interfaces

Ident.	OpenTravel Message	Distinct complex types	Shared complex types
AIR01	OTA_AirAvailRQ/RS	15	6
AIR02	OTA_AirBookRQ/RS	62	6
AIR03	OTA_AirBookModifyRQ	12	1
AIR04	OTA_AirCheckInRQ/RS	68	10
AIR05	OTA_AirDemandTicketRQ/RS	27	6
AIR06	OTA_AirDetailsRQ/RS	19	9
AIR07	OTA_AirFareDisplayRQ/RS	41	7
AIR08	OTA_AirFlifoRQ/RS	23	9
AIR09	OTA_AirPriceRQ/RS	48	8
AIR10	OTA_AirRulesRQ/RS	8	4
AIR11	OTA_AirScheduleRQ/RS	15	6
AIR12	OTA_AirSeatMapRQ/RS	30	6

Table 2: Detailed analysis of operation interfaces, source: author

As we can see from the Table 2, the interface AIR04 with messages OTA_AirCheckInRQ and OTA_AirCheckInRS uses the most of the shared complex types, 10. Another two interfaces with high number of used shared complex types are AIR06 and AIR08 with 9.

6.3.3 Experiment

Now we will update the underlying schema and will check how the metric will reflect our change.

We will use *EquipmentType complex* type for our experiment, and we will replace it with a anonymous complex type type, removing the stamp coupling from the interface. This element type appears in *AirDetailsRS* and *AirFlifoRS* messages.

Now if we run the metric again, we will have following results

DCI =	4.3333335,
ANCT =	30.666666,
NDCI =	0.14130436.

As we can see, the results of DCI have slightly changed, which reflects our change in the schema definition.

6.4 Summary

As was demonstrated in this chapter, the tool for measuring DCI metric can be used as a guide for potential problems identification in service interface design.

However, tool alone in not enough to locate these problems. Various questions need to be answered by the domain expert, for example whether given complex type represents a business entity or whether it is just used by the infrastructure. This is important to consider, as stamp coupling is a problem only in case of complex types representing business entities.

Various strategies can be used for removing stamp coupling. We have demonstrated how replacing shared complex types with anonymous (non shared) is reflected by the metric results. Similar results could be achieved by optimizing service granularity.

7 Summary and conclusion

The design of any software product possesses a number of properties that can be assessed by measuring the structure of the design artifacts using software metrics. Since these properties have causal impact on the quality characteristics of the final software product, using these software metrics can be very beneficial.

In this thesis we have summarized how similar approach can be used for evaluating quality of service design and we have reviewed available metrics. As we have seen, there are metrics available for measuring structural properties from all different angles. An intensive empirical evaluation is required in order to validate practical usefulness of those metrics.

In order to use these metrics effectively during the Software Development Lifecycle, it is necessary to have tools that can measure structural properties according to the metrics automatically. Moreover, metrics alone cannot identify the problem. They can act only as a guide, but various consideration need to be made by a domain expert before the actual problem can by determined. Software can measure only those properties, which can be directly quantified. Unfortunately, lot of important factors are only of conceptual notion, thus very hard to quantify.

We have demonstrated how such tool can be implemented and which considerations need to be made during the service oriented design quality evaluation.

However, even with all the disadvantages, software metrics have their rightful place in the software development quality assurance process.

8 References

8.1 Articles

[1] Feuerlicht, G., Enterprise SOA: What are the benefits and challenges, System Integration, Prague, Czech Republic, June 2006 in Proceedings of System Information 2006, ed Pour J., Vorisek, J., PUE, Prague, Czech Republic, p. 36-40.

[2] Feuerlicht, G., and Wijayaweera, A. Determinants of Service Reusability, in the Proceedings of the 6th International Conference on Software Methodologies, Tools and Techniques, SoMet 07, November 7-9, 2007, Rome, Italy, IOS Press, Volume 161, Pages: 467-474, 2007

[3] Feuerlicht G., Simple Metric for Assessing Quality of Service Design, to be published in the proceedings of Six International Workshop on Engineering Service-Oriented Applications (WESOA 2010), ICSOC 2010, San Francisco, December 7, 2010

[4] Sindhgatta, R., B. Sengupta, and K. Ponnalagu, Measuring the Quality of Service Oriented Design, in Service-Oriented Computing, L. Baresi, C.-H. Chi, and J. Suzuki, Editors. Springer Berlin / Heidelberg. p. 485-499. 2009.

[5] Perepletchikov M., Ryan C., and Frampton K., Cohesion Metrics for Predicting Maintainability of Service-Oriented Software (QSIC). In Proceedings of the 7th IEEE International Conference on Quality Software, pages 328–335, 2007

[6] Eder J., Kappel G., Schrefl M., Coupling and Cohesion in Object-Oriented Systems, Technical Report, University of Klagenfurt, 1994.

[7] Stevens, W. P., Myers, G. J. and Constantine, L. L., Structured Design, IBM Systems Journal, Vol. 13, No. 2, May 1974.

[8] Visser J., Structure metrics for XML schema. Departamento de Informatica Universidade do Minho Braga, Portugal, Proceedings of XATA 2006.

[9] Athanasopoulos D., Zarras A. Fine-grained Metrics of Cohesion Lack for Service Interfaces (Industry Track). 9th International Conference on Web Services (ICWS), July 2011

[10] Pham Thi Quynh, Huynh Quyet Thang: Dynamic Coupling Metrics for Service-Oriented Software. Journal of Science and Technology, ISSN 0868-3980, No. 73, 2009

[11] Haesen R., Snoeck M., Lemahieu W., Poelmans S., On the Definition of Service Granularity and Its Architectural Impact, Proceedings of the 20th international conference on Advanced Information Systems Engineering, June 16-20, 2008

8.2 Books

[11] Erl, T.,SOA: Principles of Service Design. Prentice Hall, 2008. ISBN 0-13-234482-3

[12] Erl, T., SOA Design Patterns. Prentice Hall, 2009. ISBN 0-13-613516-1

[13] Erl, T., Service-Oriented Architecture: Concepts, Technology, and Design, Prentice Hall, 2005, ISBN: 0-13-185858-0

8.3 Other resources

[14] Wiersma R., Finding an optimum in service granularity, Master Thesis. HU University of Applied Sciences, Faculty of Science Engineering 2010. Available online: http://hbo-kennisbank.uvt.nl/cgi/hu/show.cgi?fid=28420

[15] Perepletchikov, M., Software Design Metrics for Predicting Maintainability of Service-Oriented Software. Doctoral thesis, School of Computer Science and Information Technology College of Science, Engineering and Health RMIT University. 2009

[16] ISO/IEC, ISO/IEC 9126-1:2001 Software Engineering: Product quality - Quality model, International Standards Organisation, Geneva 2001

[17] OTA. OTA Specifications. 2010 [cited 6 May 2010]; Available from: http://www.opentravel.org/Specifications/Default.aspx.